PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

REDES INTRA-CHIP PARAMETRIZÁVEIS COM INTERFACE PADRÃO PARA SÍNTESE EM HARDWARE

por

LUCIANO COPELLO OST

Dissertação de mestrado submetida como requisito parcial
à obtenção do grau de Mestre em Ciência da Computação.

Prof. Dr. Fernando Gehm Moraes
Orientador

Porto Alegre, Março de 2004.

RESUMO

A evolução tecnológica tem permitido aumentar a complexidade dos projetos, e por conseqüência o número de núcleos de propriedade intelectual em um mesmo circuito integrado. Este acréscimo de núcleos conduz à pesquisa de novas estruturas de interconexão intra-chip que devem atender as necessidades de comunicação dos futuros SoCs: escalabilidade, reusabilidade e paralelismo. Uma destas estruturas de interconexão são as redes intra-chip, ou NoCs. As redes intra-chip utilizam conceitos de redes de interconexão de computadores paralelos e redes de computadores. Para auxiliar no projeto de SoCs a reusabilidade das estruturas de interconexão e dos módulos de hardware é indispensável. Para aumentar a reusabilidade de projeto, as interfaces externas dos núcleos e as interfaces das estruturas de interconexão devem ser padronizadas. Neste cenário, o presente trabalho concentra-se no desenvolvimento de redes intra-chip parametrizáveis com interface padrão OCP para síntese em hardware. A maior contribuição desse trabalho consiste na unificação destas frentes de estudo, NoCs e interfaces padrão, através do desenvolvimento da ferramenta Maia, a qual automatiza a geração de redes intra-chip com interface padrão OCP.

Palavras-chave: Redes intra-chip, interface de rede, protocolo OCP.

ABSTRACT

The technology evolution allows the increasing of design complexity and consequently the number of IP cores in the same integrated circuit. This increasing number of IP cores drives the research of new intra-chip interconnection infrastructure, which must allow the communication requirements for future SoCs: reusability, scalability, and parallelism. A network on chip draws on concepts inherited from distributed systems and computer networks subject areas to interconnect IP cores in a structured and scalable way. The reusability of intra-chip interconnection infrastructure and IP cores are seen as a needed feature to enable future SoC designs. However, for increasing the reusability of the design, the external interfaces of the IP cores and the intra-chip interconnection infrastructure must be standard. In this context, the present work focuses the implementation of parametric NoCs with standard interfaces using OCP protocol to synthesize on FPGA. The main contribution of this work is the unification of the studies (NoC and OCP standard interface) using the Maia tool which automates the generation of NoCs with OCP standard interface.

Keywords: network on chip, network interface, OCP protocol.

AGRADECIMENTOS

Agradeço ao CNPq pelo suporte financeiro que permitiu a realização deste trabalho. Hummmmm lembrei... bolsa? Ai meu Deus... deu dor de cabeça essa estória. Vamos aos fatos: inicialmente, meus estudos eram financiados pela PARKS mas em função de alguns acontecimentos essa bolsa foi para o espaço. Aproveito o momento para agradecer a PUC pelo amparo financeiro recebido durante os meses de setembro, outubro e novembro de 2002 (período que estive sem bolsa). Ainda dentro do escopo dessa história, duas pessoas merecem o meu sincero VALEU!!! Moraes e Ney, pela luta que vocês abraçaram para conseguir a bolsa para o super OST ☺. Espero que vocês não estejam arrependidos.

Às pessoas que contribuem e muito para minha existência, meus queridos amigos (em ordem alfabética para não rolar ciúmes): Ana Paula Felix, Cecília; Fabiano Brites; Ricardo Fuchs; Daniel Gonzalez; Juliano M. Santos; Leandro Galvão; Marcelo Menezes; Marcio Cristiano. O meu VALEU A TODOS!!!! Devo destacar a ajuda que recebi do Ricardo (Fuca), da Cecília Giordano (Ce, não vou colocar o apelido para ela não ficar braba) e do Marcelo (Guiso). Que ajuda foi essa? Digamos que eu vivia na casa deles para usar a Internet (banda larga) e os micros pois o meu estava perere. Ou seja, eles abdicavam da privacidade para agüentar o super OST (bem que isso é um privilégio, não é ☺?). Essa ajuda foi indispensável para a conclusão dessa dissertação. Não posso esquecer de duas situações: (*i*) a aparição do Fabiano (toma colorado) na defesa do PEP, não esqueci; (*ii*) a recepção e a ajuda do Marcio (toma colorado) quando estive em SP para participar do SBCCI'03. Ana Paula (anjinha), muito obrigado pela força és uma amiga e tanto. Obrigado por ligar/escrever quase todos os dias "apenas" para dizer: "tenha um bom dia de trabalho". Juliano e Leandro, valeu pelos conselhos e discussões que tínhamos sobre o meio acadêmico. Estes conselhos foram e serão importantes. Daniel, não poderia deixar de mencioná-lo, valeu por tudo: principalmente por ter me agüentado durante um ano (quando dividíamos o apto.). E é claro, pelas partidas de futebol que ajudavam a esfriar a cabeça cansada de tanto trabalho ☺!!!.

Aos meus amigos do mestrado e membros do GAPH (Edson Moreno, Eduardo Brião, Ewerson Carvalho, que me sacanearam pra burro) agradeço pela ótima convivência que me proporcionaram. Seria muuuuuito mais difícil sem vocês!!!! Vocês conseguem imaginar o mestrado sem o chimarrão, as cervejadas, as partidas de futebol (que eu dei aula ☺) e as jantas (claro, com as minhas piadas ☺ hehehehehehe). Ressalto aqui alguns nomes: João Carlos Ribeiro, Leonardo Brenner, Osmar Marchi dos Santos e Afonso Sales. Vocês quatro foram show de bola (lembram dos jogos do Grêmio?). Em especial, agradeço ao João Carlos que a mais de seis anos contribui com seu companheirismo. Alessandro Noriaki (mas é pilha esse japa)... sim... o japa mais alegre que eu conheci. Você é um cara legal, difícil não rir do seu lado. Mas também não esqueci o quanto você me sacaneou (toma corinthiano, vou te furar o olho ☺). Não poderia esquecer do César Augusto Missio Marcon (fala Marcon). Não esqueci dos seus conselhos e, principalmente, das observações proferidas sobre a minha pessoa. Você é um cara fantástico (e sacana também ☺)!!! Cabe ainda lembrar, do Leandro Soares Indrusiak, meu ex-orientador mas ainda conselheiro. Lembro até hoje das suas palavras no SBCCI de 2002: "Entra no eixo guri, já ta na hora...". Mais uma vez Valeu!!!

À Aline Vieira (fala garotinha), sem o teu esforço e dedicação o resultado não seria o mesmo. O toque feminino ☺ que acrescentastes foi essencial. Além da Aline, cito as contribuições do José Carlos e do Everton Carara. Ahhhh, ao Alexandre Amory e ao Leandro Möller também (dois caras que sabem muito e que são super humildes)....Valeuuuuuuuuu...

Ao Ney apesar das inúmeras definições “lapidadas” ao super Ost, devo afirmar que foi satisfatório ter convivido esses dois anos contigo. Aprendi e muito com suas dicas, correções e, principalmente, com o seu perfeccionismo perfeito (não vou definir e nem colocar referência viu estamos nos agradecimentos). Certamente levarei uma porcentagem disso!!! Valeu pela força....

Ao Moraes, sim e agora? O que escrever? Inicialmente, você vai para o Céu!!! Foram dois anos de convivência, onde me ajudastes a construir o conhecimento técnico indispensável para realização dessa dissertação. Porém, este conhecimento não é o ensinamento mais valioso que me passastes. Ressalto duas coisas: (*i*) “...faça as coisas certas como devem ser...”; (*ii*) “...não podemos esquecer de lado pessoal, isso também é importante...”. Tu és um exemplo a ser seguido, tanto como pesquisador quanto pessoa. Acho que não preciso escrever mais nada. Apreendi muito contigo nesses dois anos. Logo, você ganha mais três VALEU VALEU VALEU e de lambuja um YUPI ☺!!!!! Eu gostaria de mencionar uma frase do Albert Einstein mas como não lembro dela direito...

Aos meus pais (papai Eugenio e mamãe Bernadett) que sempre me deram força e incentivo. Vocês já sabem o quanto sou grato a vocês!!! Tenho muito tempo para incomodar vocês dois. Hoje sou o que sou graças aos dois.... mas vocês tem sorte hein ☺!!!!

À Lílian C. Ost, minha maninha querida (a Dot), que agüentou o meu humor ☹ quando chegava em casa cansado. Valeu Lilica!!! Te adoro!!!

Valeu!!! Valeu mesmo!!! YUPI YUPI YUPI!!! 

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

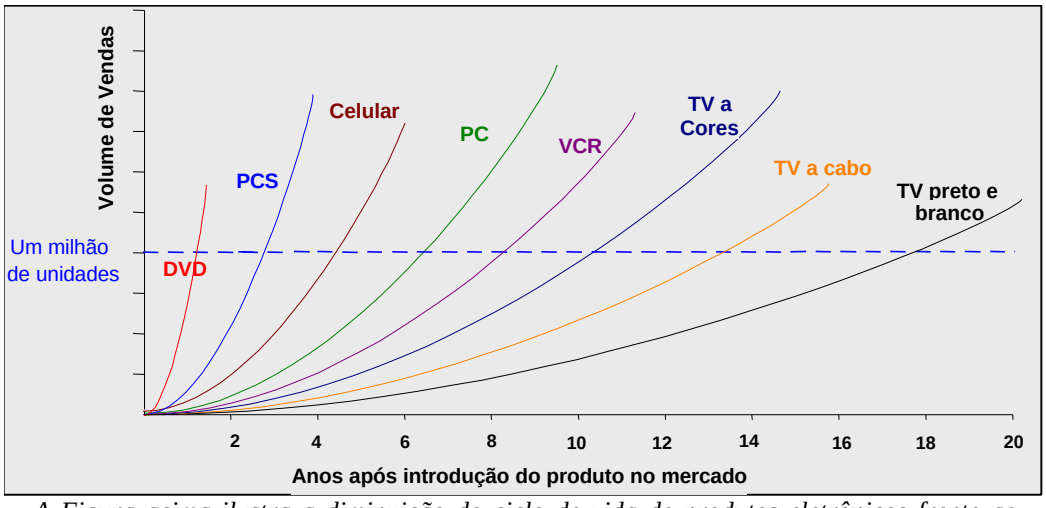
CI	*Circuito Integrado*
DSP	*Digital Signal Processor*
DVD	*Digital Video Disk*
FIFO	*First in First Out*
FPGA	*Field Programmable Gate Array*
HDL	*Hardware Description Language*
IP	*Intellectual Property*
MEMS	*Micro-Electro-Mechanical Systems*
NoC	*Network on a Chip*
NI	*Network Interface*
IR	*Interface de Rede*
OCP	*Open Core Protocol*
QoS	*Quality of Service*
SIA	*Semiconductor Industry Association*
SoC	*System on a Chip*
STL	*Socket Transaction Language*
VCI	*Virtual Core Interface*

1 INTRODUÇÃO

O aumento do número de transistores e da freqüência de operação, o curto tempo de projeto e a redução do ciclo de vida dos produtos eletrônicos caracterizam o cenário atual da indústria de semicondutores. Em 1980, a maioria dos circuitos integrados (CIs) ditos complexos era composta por dezenas de milhares de transistores. Atualmente, é possível encontrar CIs que contenham dezenas de milhões de transistores [REI00]. Contudo, estima-se que até 2012 existam CIs contendo 4 bilhões de transistores, operando a uma freqüência de 10 GHz [ITR02]. Esse avanço tecnológico permite implementar sistemas computacionais completos em um único CI, denominados SoCs (do inglês, *Systems on a Chip*) [BER00][BER01].

SoCs podem ser compostos por processadores, memórias, módulos dedicados em hardware para realização de funções específicas, módulos de software e inclusive tecnologias mais recentes como MEMs (do inglês, *Micro-Electro-Mechanical Systems*) integrados no mesmo CI [MAR01] [JUN01][ITR02]. A heterogeneidade apresentada por esses sistemas, ou seja, a possibilidade de combinar diferentes tecnologias no mesmo CI, tem sido aproveitada em diversos setores da indústria. Tal fato é evidenciado pelo sucesso de produtos eletrônicos como telefones celulares, PDAs, aparelhos de jogos eletrônicos, aparelhos de DVDs, máquinas fotográficas digitais (integração de sensores ópticos com lógica digital e analógica), entre outros produtos [BEN01][RUN00][SIA99]. Porém, combinar diferentes tecnologias no mesmo CI é um processo difícil, que implica o aumento de tempo e custo de projeto de novos produtos. Por outro lado, existe a necessidade da diminuição no tempo de lançamento do produto no mercado (*time-to-market*), o que vêm a contribuir para o domínio de mercado e aumento dos lucros.

Ambos os fatores, evolução tecnológica e o curto *time-to-market* possuem grande efeito na diminuição do ciclo de vida dos produtos. Segundo Bergamaschi [BER02], a diminuição do ciclo de vida dos produtos pode ser evidenciada pelo tempo que um dado produto leva para ter o seu volume de vendas igual a 1 milhão de unidades, como ilustrado na Figura 1.



A Figura acima ilustra a diminuição do ciclo de vida de produtos eletrônicos frente ao volume de vendas após a introdução dos mesmos ao mercado. A mesma figura mostra que um televisor a cores levou aproximadamente 10 anos para vender 1 milhão de unidades, enquanto, que em um ano vendeu-se o mesmo volume de aparelhos de DVD.

Figura 1 - Diminuição do ciclo de vida dos produtos [BER02].

Sendo assim, é possível constatar que existe uma grande pressão em relação à diminuição do tempo de desenvolvimento e de lançamento de um produto ao mercado antes que o mesmo já esteja ultrapassado tecnologicamente. Dentro deste cenário, as companhias que projetam e vendem SoCs priorizam em seus projetos a utilização de núcleos de propriedade intelectual, ou simplesmente núcleos (do inglês, *IP cores*), a fim de aumentar a produtividade, minimizando o tempo de desenvolvimento, e conseqüentemente, o *time-to-market* de seus produtos [BER00].

Apesar das vantagens inerentes à utilização de núcleos no projeto de SoCs, identificam-se quatro grandes problemas que precisam ser resolvidos para facilitar tal prática. Esses problemas são: (*i*) integração dos núcleos entre si para compor o SoC; (*ii*) escolha das linguagens para descrever os núcleos e o SoC; (*iii*) proteção à propriedade intelectual; (*iv*) teste dos núcleos e do SoC [BER01]. O escopo do presente trabalho de mestrado limita-se ao primeiro desafio, ou seja, a integração de núcleos IPs em SoCs.

1.1 Desafios para o projeto de SoCs

1.1.1 Integração de núcleos

O problema de integração de núcleos pode ser abordado a partir de duas questões: (*i*) as estruturas de interconexão utilizadas atualmente para integrar núcleos suportarão os requisitos de desempenho e de escalabilidade dos futuros SoCs?; (*ii*) como integrar núcleos com interfaces externas diferentes à estrutura de interconexão adotada pelo SoC alvo?

Duas estruturas de interconexão são usualmente utilizadas para interconectar núcleos em um SoC: *fios ponto-a-ponto dedicados* e *barramentos*, simples ou hierárquicos [KUM03]. Diversos autores prevêem que as atuais estruturas de interconexão tornar-se-ão fatores limitantes para grandes projetos, em função do aumento da complexidade dos sistemas e do aumento dos requisitos de largura de banda. Uma possível solução para a integração de núcleos são as redes intra-chip (*Network on a Chip* - NoC) [BEN01][DAL01][SAA02][GUE99][GUE00][KUM02][SGR01]. Nesta abordagem, um SoC pode ser visto como uma micro-rede de componentes, onde a rede é o meio responsável pela comunicação entre os componentes do sistema, ou seja, os núcleos [BEN02]. Esta forma de interconexão é discutida no Capítulo 2.

Núcleos são desenvolvidos para um protocolo de comunicação, proprietário ou padrão. Durante o desenvolvimento de um dado SoC, os núcleos selecionados para compor o mesmo podem ser provenientes de diferentes provedores, os quais podem adotar protocolos distintos daquele associado à estrutura de interconexão que está sendo utilizado no projeto do SoC. Logo, para inserir tais núcleos no SoC, existe a necessidade de adição de lógica de cola (do inglês, *glue logic*) que permite a comunicação entre o núcleo e o restante do circuito [BER01]. Essa lógica é denominada invólucro (do inglês, *wrapper*) ou adaptador. A necessidade de adaptação de núcleos reduz a reusabilidade do mesmo, e aumenta o tempo de projeto do SoC [ITR02]. A Figura 2, ilustra o problema de interconectar núcleos com interfaces diferentes à estrutura de interconexão de um SoC.

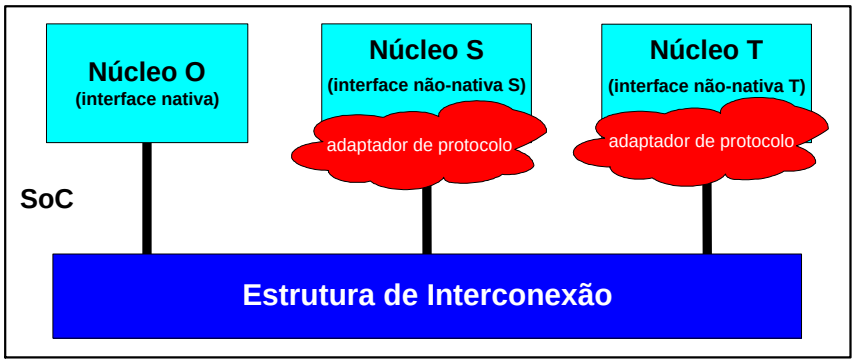


Figura 2 - Interconexão de núcleos com interfaces distintas.

Uma alternativa para evitar a inserção de lógica de cola é padronizar a(s) interface(s) entre o(s) núcleos e a estrutura de interconexão adotada [BIR99]. Dois padrões de interfaces bastantes difundidos são o VCI da VSIA (*Virtual Socket Interface Alliance*) [VSI02] e o OCP da *Sonics* [OCP02][OCP02a][OCP02b]. Esta abordagem não muda a forma de desenvolver núcleos, pois estes continuam sendo desenvolvidos para um dado protocolo. O que muda é que esse protocolo é de domínio público, e aceito pela indústria como um padrão, como foi o padrão PCI para os fabricantes de microcomputadores. Assim, a reusabilidade dos núcleos é maior e o tempo de projeto diminui, pois a integração de núcleos passa a ser um processo simplificado. O tema de utilização de interfaces padrão é discutido no Capítulo 3 deste trabalho.

1.1.2 Linguagens para Descrição de SoCs

O projeto concorrente de hardware e software é uma importante característica dos SoCs. O procedimento usual para o projeto destes sistemas é utilizar uma dada linguagem para descrever os módulos de hardware e outra linguagem para descrever os procedimentos de software. Linguagens de descrição de hardware, tais como Verilog [THO91] e VHDL [PER98], têm como características comuns hierarquia (descrição estrutural com utilização de componentes), concorrência entre processos e temporização. Linguagens para descrição de software, como C ou C++, são baseadas em um modelo de execução seqüencial, adaptadas para a execução em processadores de propósito geral. Estas linguagens para descrição de software geralmente não têm suporte para modelar concorrência e temporização.

Utilizar duas linguagens distintas, com modelos de computação também distintos, para a modelagem do SoC traz as seguintes desvantagens: (*i*) a partição do sistema já está definida, não havendo flexibilidade para mover núcleos de hardware para software e vice-versa; (*ii*) esta rigidez impede uma exploração do espaço de soluções possíveis para o sistema em desenvolvimento; (*iii*) a validação do sistema requer técnicas de co-simulação, além de ser efetuada em um nível de abstração muito próximo da implementação final (nível de ciclo), o que pode tornar a validação uma tarefa muito custosa em termos de tempo de CPU.

Em função destas características é desejável que o projeto de SoC seja feito utilizando uma única linguagem, que dê suporte aos modelos de computação utilizados tanto para modelagem de hardware quanto para modelagem de software. O uso de uma única linguagem traz como benefícios a possível exploração do espaço de soluções do projeto e um processo de validação simplificado em relação ao processo baseado em co-simulação.

Exemplos de linguagens utilizadas para modelagem concorrente de hardware e software são *SystemC* [SYS03], *SystemVerilog* [THO91] e *SpecC* [GAJ00]. Estas linguagens são derivadas de linguagens de descrição de software, acrescentando-se bibliotecas que simulam as características de paralelismo e temporização. A descrição de sistemas computacionais através destas linguagens permite também aumentar o nível de abstração de projeto. A elevação dos níveis de abstração de projeto para os chamados níveis sistêmicos permite que detalhes de baixo nível sejam abstraídos, tornando mais fácil descrever a funcionalidade de cada núcleo e suas interconexões.

A dissertação do aluno Edson Ifarraguirre Moreno [MOR03b], desenvolvida em paralelo com este trabalho, tem por foco a modelagem e avaliação de desempenho de redes intra-chip no nível de abstração transacional TLM (do inglês, *Transaction Level Modeling*), utilizando a linguagem SystemC. Seu trabalho destaca as vantagens acima citadas, exploração do espaço de soluções e redução do tempo de validação. Apesar das vantagens da utilização de SystemC, não há ainda ferramentas para síntese automática do nível TLM para o nível RTL (do inglês, *Register Transfer Level*), ponto de partida para a síntese de hardware. Não é objetivo deste trabalho utilizar níveis abstratos de descrição. Toda a modelagem dos sistemas será feita em VHDL, no nível RTL de abstração.

1.1.3 Proteção da Propriedade Intelectual

Como um provedor de núcleos (hardware ou software) pode fornecer a um projetista um dado núcleo, sem entretanto fornecer detalhes que comprometam a sua propriedade intelectual?

Este é um desafio de solução complexa, pois o projetista do núcleo deseja manter a implementação deste fechada, pois isto representa o seu conhecimento - propriedade intelectual. O usuário do núcleo deseja informações deste, pois necessita avaliar parâmetros como área, velocidade, potência e testabilidade para decidir se irá comprar ou não o mesmo e utilizá-lo em seu projeto. A compra de um dado núcleo pode representar investimento na ordem de milhares de dólares [ITR02][RUN00].

Diversas técnicas já foram propostas para proteção à propriedade intelectual de núcleos, como utilização de descrições de hardware criptografadas [DAL00], simulação remota (o usuário do núcleo instancia o núcleo desejado, porém a simulação do mesmo ocorre no servidor do provedor do núcleo) [DAL00] e modelos comportamentais dos núcleos.

A terceira alternativa, modelos comportamentais dos núcleos, mostra-se hoje a mais promissora, em função do avanço das linguagens sistêmicas, como SystemC. Caso a descrição SystemC TLM forneça uma estimativa precisa dos parâmetros do núcleo, a integração deste ao sistema é feita neste nível de abstração, permitindo ao projetista definir como será o seu sistema. Uma vez definido quais núcleos serão utilizados, o projetista adquire a descrição RTL do núcleo TLM utilizado para modelagem do sistema. Este processo traz os seguintes benefícios: (*i*) efetiva proteção à propriedade intelectual, uma vez que a estimativa dos parâmetros do núcleo é feita para um processo de síntese desconhecido do projetista do SoC; (*ii*) redução do tempo de projeto, através do reuso de núcleos.

1.1.4 Teste

Na maioria dos casos, exceto para núcleos descritos na forma de código fonte (*soft cores*), os usuários de núcleos possuem pouco conhecimento do conteúdo interno dos mesmos, tratando-os como caixa preta. O usuário tem acesso apenas à fronteira externa dos núcleos instanciados.

Os desenvolvedores dos núcleos definem o método de teste, sem o conhecimento de onde o núcleo será utilizado. Como resultado, o projetista do núcleo pode ou não fornecer um método de teste com a qualidade adequada. Se a cobertura é muito fraca, a qualidade do sistema pode ser comprometida. Por outro lado, se for muito alta, o custo de teste pode aumentar (tempo e requisitos de memória).

Visando simplificar a interface entre os provedores de núcleos, que definem o teste individual dos mesmos, e os projetistas de SoCs, que utilizam os núcleos e devem testar o sistema completo, a proposta de padrão IEEE P1500 define regras para o teste de SoCs.

A proposta de padrão P1500 [IEE02a] está focada na padronização da interface entre usuário e provedores de núcleos. Esta padronização atende: (*i*) uma linguagem padrão capaz de expressar todas as informações relacionadas a teste que devem ser transferidas para o usuário do núcleo; (*ii*) uma lógica envoltória de teste padrão configurável, que facilite a integração de núcleos no sistema. P1500 não cobre: (*i*) método de teste interno dos núcleos; (*ii*) integração do teste do sistema; (*iii*) otimizações do teste do sistema; (*iv*) mecanismos de acesso ao teste; (*v*) fonte e receptor de teste. Estes pontos não cobertos são responsabilidade dos provedores e do usuário de núcleos, e sua padronização não é viável devido às diferentes restrições dos núcleos e sistemas.

O tema de pesquisa em teste de SoCs está contemplado na dissertação do aluno Alexandre Amory [AMO02], trabalho este que esta sendo continuado em seu doutorado.

1.2 Motivação

As estruturas de interconexão atualmente utilizadas podem se tornar o gargalo no projeto dos futuros SoCs [KUM03], onde dezenas ou centenas de núcleos comporão um determinado sistema computacional. Dessa forma, a pesquisa em redes intra-chip mostra-se relevante tanto para a comunidade acadêmica quanto para a indústria.

A pesquisa e o desenvolvimento de técnicas para a integração de núcleos de forma "*plug-and-play*" (possibilidade de conectar núcleos de hardware distintos e obter funcionamento correto com o restante do sistema sem adição de lógica adicional [BIR99]), tornando a estrutura de interconexão transparente para o projetista, representa a principal motivação para a realização desse trabalho.

1.3 Objetivos do Trabalho

O objetivo *principal* deste trabalho é dominar a tecnologia de interconexão de núcleos através da utilização de redes intra-chip, e a utilização de interfaces padronizadas para a comunicação entre os núcleos. A contribuição maior do presente trabalho, resultado deste objetivo, é o desenvolvimento de *redes-IP*[1] parametrizáveis com interface padrão OCP para síntese em hardware.

O conceito de *redes-IP* refere-se a redes intra-chip gerada automaticamente a partir de uma ferramenta, em função das restrições da aplicação alvo, com interface padronizada. A rede intra-chip utilizada para este trabalho é denominada HERMES [MOR03], desenvolvida no grupo de pesquisa GAPH. O protocolo de comunicação utilizado na interface da rede é denominado OCP. A Figura 3 ilustra uma *rede-IP*.

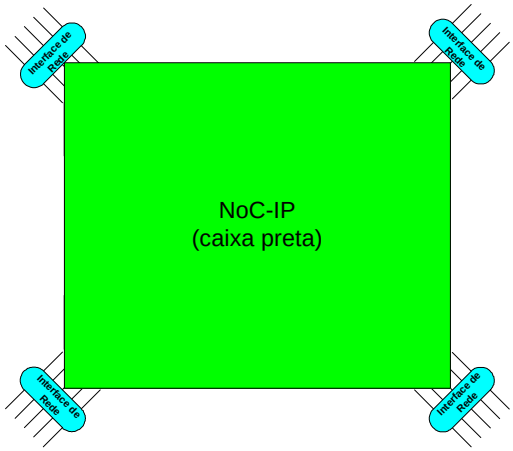


Figura 3 - Rede-IP com 4 interfaces de rede baseadas no protocolo OCP.

Os objetivos *específicos* deste trabalho são:

- Estender a rede HERMES para as topologias anel, *torus* e *torus* dobrado.
- Reduzir a utilização de área da rede através da remoção dos *buffers*[2] não utilizados. Esta otimização é importante, dado que a fila é o componente que mais consome área na chave.
- Desenvolver as interfaces de rede OCP (mestre, escravo, mestre-escravo) para a rede HERMES.
- Desenvolver uma ferramenta para geração da rede intra-chip HERMES, com interface OCP, em função dos parâmetros do projetista. Além da rede-IP, a ferramenta deve gerar os arquivos de configuração utilizados pela ferramenta *CoreCreator*.
- Apresentar pelo menos um estudo de caso de desenvolvimento de SoC com a rede HERMES-OCP.
- Prototipar o estudo de caso em FPGA.

[1] O termo rede-IP refere-se a uma rede intra-chip como núcleo IP.
[2] No contexto deste trabalho, será utilizado o termo fila como sinônimo.

1.4 Organização do Documento

Este trabalho está organizado como segue. No Capítulo 2 são discutidos conceitos de SoCs, com ênfase às estruturas de interconexão. Neste Capítulo são apresentados conceitos e terminologias de redes intra-chip. Este Capítulo também inclui uma revisão do estado-da-arte de redes intra-chip.

O Capítulo 3 aborda a questão de interfaces padrão, enfatizando o padrão OCP, escolha do presente trabalho. Um exemplo detalhado referente ao desenvolvimento e validação de um núcleo com interface padrão também é descrito.

O Capítulo 4 apresenta a rede intra-chip HERMES e a *primeira contribuição* deste trabalho: o desenvolvimento de interfaces de rede OCP para a rede HERMES. Resultados referentes à implementação e validação funcional (software e hardware) também são apresentados nesse Capítulo.

O Capítulo 5 apresenta a *segunda contribuição* deste trabalho, que consiste no desenvolvimento de uma ferramenta para geração automática de redes-IP com interface OCP, segundo parâmetros definidos pelo usuário. Esta ferramenta também gera tráfego para a validação da rede.

O Capítulo 6 apresenta um estudo de caso de projeto de SoC, que aborda: (*i*) o empacotamento dos núcleos frente ao protocolo OCP; (*ii*) certificação dos núcleos; (*iii*) geração da rede intra-chip a partir da ferramenta citada acima; (*iv*) integração dos núcleos à rede gerada.

Finalmente, o Capítulo 7 apresenta as considerações finais sobre este trabalho e direções para trabalhos futuros.

2 REDES INTRA-CHIP

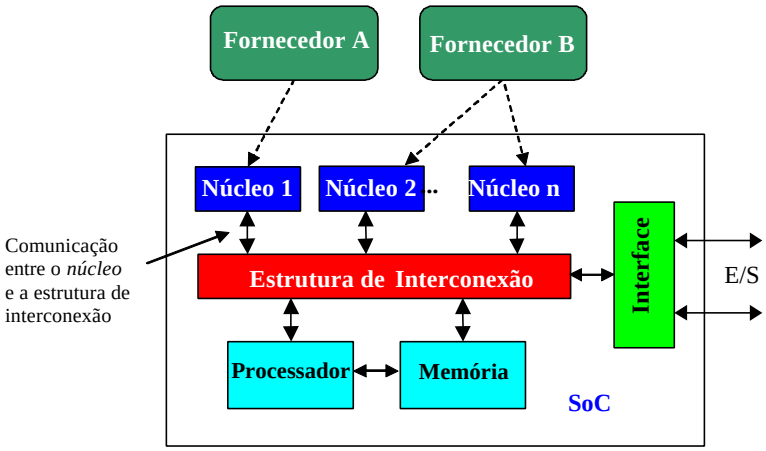
Este Capítulo apresenta os conceitos básicos de SoCs e como os núcleos são interconectados entre si no mesmo. A forma de interconexão por redes intra-chip é estudada em maior profundidade pois esta atende aos requisitos de desempenho e escalabilidade dos futuros SoCs. Ainda nesse Capítulo apresenta-se o estado da arte em redes intra-chip e considerações sobre a integração de núcleos a essa estrutura de interconexão.

2.1 SoCs

Como descrito anteriormente, o mercado de semicondutores é caracterizado por produtos eletrônicos cada vez mais complexos, com reduzido tempo de vida. Com isso, tornou-se comum desenvolver SoCs a partir de núcleos heterogêneos. A agência de pesquisa In-Stars estima que o mercado mundial de SoCs deve crescer de $20 bilhões no ano de 2000 (20% do mercado de CIs atual) para $60 bilhões em 2004, ou seja 40% do mercado, representando um crescimento aproximado de 31% ao ano.

A Associação das Indústrias de Semicondutores [SIA99] estima que em 2005 90% da área dos circuitos integrados seja preenchida por núcleos. Logo, é possível inferir que a produção de SoCs depende da criação e da validação de núcleos que possam ser reutilizados em projetos distintos. Estima-se que criar um núcleo que possa ser reutilizado por outros projetistas é substancialmente mais difícil (por um fator estimado entre 2 e 5 vezes maior) que desenvolvê-lo para um único projeto [ITR02].

Como ilustrado na Figura 4 [MAD97], um SoC é composto por núcleos não programáveis, processadores e memórias que comunicam-se através de uma estrutura de interconexão e interfaces com o mundo externo.



A arquitetura acima é composta por núcleos IP. Estes núcleos podem ser módulos de telecomunicação, processadores DSP (Digital Signal Process), decodificadores de MPEG2 ou MP3, entre outros. Os núcleos provêm de fornecedores distintos (A, B) e estão integrados a uma estrutura de interconexão que pode ser um barramento ou uma rede intra-chip. A(s) interface(s) com o mundo externo (I/O) é(são) utilizada(s) para interconectar periféricos, como: porta serial, porta USB ou uma UART (Universal Assynchronous Receiver Transmiter).

Figura 4 - Arquitetura genérica de um SoC.

Gupta et al. [GUP97] definem um núcleo como um módulo de hardware pré-projetado e pré-verificado, que pode ser usado na construção de uma aplicação maior ou mais complexa em um CI. Estes núcleos podem ser classificados em três categorias: (*i*) *soft core*, (*ii*) *firm core* e (*iii*) *hard core*.

- Um *Soft Core* é a descrição de um núcleo em uma linguagem de descrição de hardware (e.g. VHDL, Verilog, SystemC). As principais vantagens apresentadas por um *soft core* são: a independência de tecnologia e a flexibilidade. É possível que o projetista/usuário modifique o mesmo, visando adequar a funcionalidade deste em prol do sistema desejado. Em contrapartida a esta flexibilidade, é responsabilidade do projetista atender às restrições temporais. Normalmente, os *soft cores* são acompanhados de *scripts* de síntese, que guiam o projetista na etapa de concepção.

- Um *firm core* é um *netlist* que apresenta menos flexibilidade em relação ao *soft core* e na maioria dos casos dependem de uma tecnologia específica (e.g. *netlist* EDIF obtido a partir de síntese lógica). As vantagens em relação aos *soft cores* são: melhor proteção da propriedade intelectual e estimativa de desempenho mais próxima da realidade.

- Um *hard core* é geralmente uma descrição de um *layout* posicionado e roteado de um ASIC/FPGA. Para atingir desempenho, baixo consumo e menor área, estes são otimizados para uma dada tecnologia. Logo, os *hard cores* garantem os tempos de propagação do núcleo (*timing*), além de proverem alta proteção à propriedade intelectual. Como conseqüência, a flexibilidade é mínima e este é fortemente dependente da tecnologia.

Os núcleos são usualmente interconectados em um SoC através de duas estruturas de interconexão: fios ponto-a-ponto dedicados e barramentos[1], simples ou hierárquicos. Antes de descrever as estruturas de interconexão citadas acima, torna-se necessário definir alguns conceitos que caracterizam as mesmas.

- *Paralelismo*: relaciona-se à possibilidade de transferência e/ou recepção de dados entre dois ou mais pares de núcleos simultaneamente;

- *Consumo de energia*: determina a quantidade de energia consumida por um determinado circuito;

- *Escalabilidade*: refere-se à capacidade de interconectar componentes adicionais à estrutura de interconexão, sem comprometimento significativo no desempenho global do sistema [KUM03];

- *Reusabilidade:* é a capacidade de utilizar uma dada estrutura de interconexão em projetos distintos. Essa estrutura deve proporcionar facilidades para que um grande número de núcleos possam trocar informações eficientemente. Isso, tanto para pares de núcleos como para comunicações concorrentes entre vários pares.

[1] Na bibliografia é possível encontrar o termo fios globais (do inglês, *global wires*).

2.1.1 Interconexão de núcleos baseada em fios ponto-a-ponto dedicados

Na abordagem baseada em fios ponto-a-ponto dedicados (do inglês, *dedicated wires*), os núcleos são interligados diretamente um ao outro, ou seja, conexão ponto-a-ponto. O desempenho oferecido por essa estrutura pode ser considerado bom, pois cada comunicação ocorre independentemente das demais. Este tipo de estrutura de interconexão é eficaz se cada núcleo tem que se comunicar com um número pequeno de núcleos. Caso pretenda-se interligar um núcleo a vários outros, o número de fios dedicados aumenta proporcionalmente ao número de núcleos, o que pode gerar o congestionamento de fios em volta do mesmo [ZEF03a]. Tal característica pode ser considerada uma limitação, à medida que projeta-se SoCs com dezenas a centenas de núcleos [ITR02][KUM03]. Outro fator limitante está no fato que o projeto deste tipo de estrutura é específico e, portanto, a reusabilidade é limitada. A Figura 5 ilustra a abordagem de interconexão descrita acima.

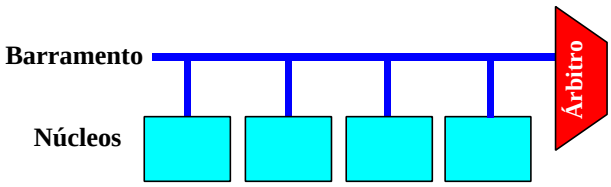
Figura 5 - Estrutura de Interconexão baseada em fios ponto-a-ponto dedicados.

2.1.2 Interconexão de núcleos baseada em barramentos compartilhados

Uma estrutura de interconexão mais reutilizável é a baseada em barramentos compartilhados (do inglês, *shared data bus*). Além da reusabilidade, a baixa área de silício e a baixa latência contribuem para que esta abordagem seja a mais utilizada para interconexão de núcleos nos SoCs atuais [GUE00].

Um barramento consiste em um conjunto de fios que conectam diferentes núcleos do SoC, e sobre o qual dados são transmitidos e recebidos. Estes núcleos podem ser classificados como mestres e/ou escravos do barramento. Um núcleo mestre é uma unidade que controla a transferência em um barramento, ou seja, pode solicitar a transmissão ou a recepção dos dados através do barramento. Por outro lado, um componente escravo é a unidade que apenas responde às solicitações desses mestres [HWA93]. Como exemplo, pode-se citar um microprocessador, atuando como mestre, e uma memória que efetua o papel de escravo do barramento. As informações são lidas ou escritas da/na memória a partir dos sinais gerados pelo microprocessador.

A maioria dos barramentos define um método de arbitragem responsável pelo controle de acesso dos núcleos mestres a si. Dentre os vários métodos citam-se o *centralizado* e o *distribuído* [HWA93]. No método centralizado, um dispositivo denominado como árbitro ou controlador de barramento é responsável pela atribuição de acesso ao barramento. Neste caso, existem sinais de requisição e de permissão. Quando o árbitro percebe uma requisição de direito de acesso, ele gera um sinal de permissão, quando for possível, ao mestre que solicitou o acesso ao barramento. No método de arbitragem descentralizado/distribuído, não há um árbitro. Uma das formas utilizadas para descentralizar a arbitragem é de delegar o monitoramento das linhas de requisição aos próprios núcleos do barramento. Desta maneira, cada núcleo sabe sua prioridade na ordem destas requisições e se podem ou não utilizar o barramento. A Figura 6, ilustra uma estrutura de interconexão baseada em barramento.



A figura acima ilustra uma arquitetura de barramento composta por um único barramento (canal de comunicação) compartilhado entre todos os núcleos sendo estes mestres e/ou escravos. Esta arquitetura possui um método de arbitragem centralizado, ou seja, existe um árbitro responsável pelo controle de acesso ao barramento.

Figura 6 - Núcleos interligados através de um barramento, com arbitragem centralizada.

Apesar de barramentos serem reutilizáveis, pode-se enumerar 3 desvantagens desta forma de interconexão: (*i*) ausência de paralelismo; (*ii*) baixa escalabilidade (*iii*) alto consumo de energia. O paralelismo inexiste em barramentos simples, pois apenas uma transação de comunicação é permitida por vez, dado que todos os núcleos compartilham o mesmo canal de comunicação. Barramentos hierárquicos podem aumentar o número de transações simultâneas, porém de uma forma limitada (duas a três transações simultâneas). A escalabilidade é limitada a dezenas de núcleos, segundo [BEN02][KUM02][GUE00]. O consumo de energia é elevado devido à existência de fios longos [BEN01]. O acréscimo de núcleos ao barramento aumenta a capacitância total do sistema, o que acaba reduzindo o desempenho elétrico.

Entre as arquiteturas de barramento intra-chip encontradas na literatura, destacam-se: *AMBA* da ARM [ARM02], *Avalon* da Altera [ALT02], *CoreConnect* da IBM [IBM02]. Geralmente, estas arquiteturas de barramento estão vinculadas à arquitetura de um processador, tal como o *AMBA* vinculado ao processador ARM [FUR00], o *CoreConnect* vinculado ao processador *PowerPC* e o Avalon vinculado ao processador *Nios*.

Diversos autores prevêem que as estruturas de interconexão citadas acima tornar-se-ão fatores limitantes para grandes projetos, em função do aumento da complexidade dos sistemas e do aumento dos requisitos de largura de banda. Uma possível solução para a integração de núcleos são as redes intra-chip.

2.2 Conceitos básicos de redes-intra chip

Como descrito em [RIJ03a], as redes intra-chip estão emergindo como uma possível solução para os problemas associados às estruturas de interconexão, devido a algumas características como: (*i*) escalabilidade da largura de banda se comparada à arquiteturas de barramento [GUE00]; (*ii*) reusabilidade; (*iii*) confiabilidade; (*iv*) eficiência em termos de consumo de energia [BEN01]. Redes intra-chip compartilham muitos conceitos com redes de interconexão de computadores paralelos e de redes locais. Existem duas diferenças básicas entre as abordagens de interconexão citadas acima: o número de componentes e a distância entre si. O número de núcleos em SoCs é inferior ao número de componentes em redes locais e a distância entre os núcleos é muito inferior aos componentes de uma rede local. Estas diferenças conduzem ao desenvolvimento de protocolos particulares às redes intra-chip.

Uma rede pode ser dividida em duas partes: os *serviços* e o *sistema de comunicação*. Rijpkema et al. [RIJ03a], descrevem alguns serviços que devem ser providos por uma rede intra-chip: (*i*) garantir a integridade de dados, ou seja, entregar estes sem serem corrompidos; (*ii*) garantir

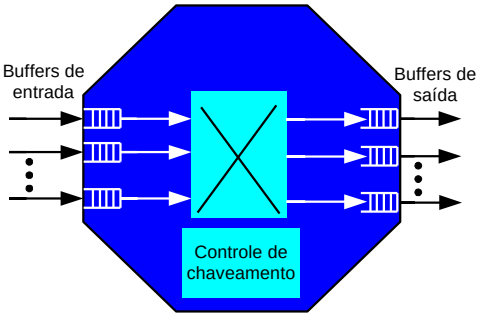
que nenhum dado vai ser perdido pela rede; (*iii*) garantir a recepção ordenada dos dados enviados; (*iv*) garantia de *throughput*, referente à quantidade de dados transferidos por unidade de tempo, (*v*) garantia de latência, referente ao tempo que uma unidade de dados demora para ser transferida do seu fonte ao seu destino.

Estes serviços pode ser classificados em [RIJ03a]: “serviços garantidos” (GT - *guaranteed throughput*) ou serviços “melhor-esforço” (BE - *best-effort services*). Serviços GT requerem reserva de recursos para o pior caso, por exemplo garantia de *throughput*, o que pode vir a ser custoso em termos de recursos de hardware (área). Os serviços BE não alocam recursos específicos, sendo mais simples de utilizar. Uma desvantagem das redes BE é a ausência de garantia de limites temporais, como latência e *throughput*.

Um sistema de comunicação deve proporcionar a transferência de informações de uma origem à um destino. Como definido por Benini [BEN02] e Guerrier [GUE00], a comunicação entre os elementos de uma rede intra-chip é baseada na transmissão de mensagem. Uma mensagem, geralmente, é composta por: (*i*) cabeçalho (*header*), (*ii*) carga útil (*payload*), contendo os dados da mensagem; (*iii*) terminador (*trailer* ou sufixo), incluindo informações utilizadas para a detecção de erros e indicar o término da mensagem. Geralmente, para transmissão divide-se a mensagem em pacotes. Um pacote é um bloco de dados que contém detalhes inerentes ao roteamento e ao seqüenciamento dos dados, mantendo uma estrutura semelhante à de uma mensagem.

Um sistema de comunicação deve permitir a troca de mensagens entre todos os núcleos conectados a estrutura da rede. A estrutura de uma rede intra-chip é basicamente um conjunto de chaves (do inglês, *switches*) conectadas entre si por canais de comunicação (meio pelo qual os dados são enviados). A forma na qual as chaves estão conectadas entre si, e os núcleos conectados às chaves, define a topologia da rede. Uma vez que a chave é o elemento principal da rede intra-chip, seu impacto na área final do SoC deve ser minimizado. Um dos fatores que mais influenciam no consumo de área são as filas para armazenamento temporário de dados. Logo, ao projetar uma chave deve existir a preocupação em termos de consumo de área e a estratégia de bufferização a ser adotada, já que este fator afeta diretamente no desempenho da rede [GUE00][RIJ03][KUM03].

Uma chave pode ser definida como um dispositivo que conecta um número de canais de entrada a um número de canais de saída [NI93]. Em outras palavras, uma chave tem a funcionalidade de transferir informações de uma de suas portas de entrada para uma de suas portas de saída. O intervalo de tempo entre a entrada e a saída de uma informação da chave é denominado de atraso de chaveamento (do inglês, *switch-delay*) [KUM03]. Geralmente, a estrutura de uma chave consiste de um módulo de controle de chaveamento, roteamento interno, e *buffers* de entrada e/ou de saída. A estrutura de uma chave genérica é ilustrada na Figura 7.

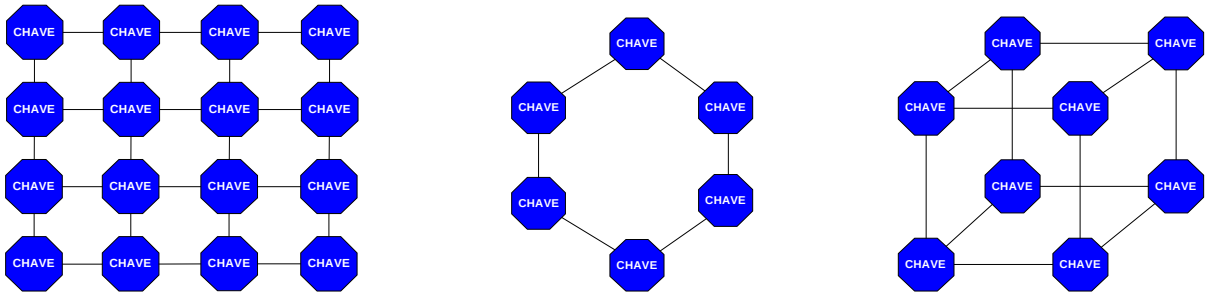


Uma chave pode possuir buffers para receber dados e buffers para armazenar dados antes de enviá-los para uma porta de saída. Deve-se notar que é possível projetar chaves que incluam buffers de entrada e/ou de saída. Além dos buffers, a chave ilustrada acima é composta por um módulo de controle do chaveamento e uma estrutura de interconexão interna, por exemplo, crossbar.

Figura 7 - Estrutura de uma chave genérica.

2.2.1 Topologias de Rede

A topologia da rede consiste na organização da mesma sob a forma de um grafo, no qual as chaves são os vértices deste e os canais de comunicação os arcos [NI93]. Em função das ligações entre as chaves, uma topologia de rede pode ser classificada como estática ou dinâmica. Se as chaves estão conectadas através de ligações fixas, de forma que entre cada duas chaves exista uma conexão dedicada, a rede de interconexão é dita estática, ou seja, ponto-a-ponto [HWA93][NI93]. Citam-se como exemplos de topologias estáticas: estrela, anel, malha, *torus* bidimensional; hipercubo de grau N. Algumas dessas topologias são ilustradas na Figura 8.



(a) malha (b) anel (c) hipercubo

Figura 8 - Exemplos de topologias estáticas. As chaves devem possuir ao menos uma interface para a conexão a um núcleo local (não ilustrado na figura).

Na interconexão de componentes com redes dinâmicas não existe uma topologia fixa que defina o padrão de interconexão da mesma. Quando uma conexão entre dois pontos se faz necessária, a rede de interconexão se adapta dinamicamente para permitir a transferência dos dados [HWA93]. Dentre as topologias de redes dinâmicas destacam-se: (*i*) matriz de chaveamento (*crossbar*), permite a conexão entre dois núcleos quaisquer, desde que estes não se encontrem já ocupados; (*ii*) redes multinível, baseiam-se na ligação de pequenas matrizes de chaveamento (normalmente de tamanho 2 x 2) em vários níveis consecutivos e conectadas de forma a reduzir a probabilidade de bloqueios entre níveis. A Figura 9 ilustra uma matriz de chaveamento 4 x 4. A matriz de chaveamento é a topologia de menor latência e melhor desempenho, porém apresenta um custo de implementação proporcional ao quadrado do número de núcleos conectados.

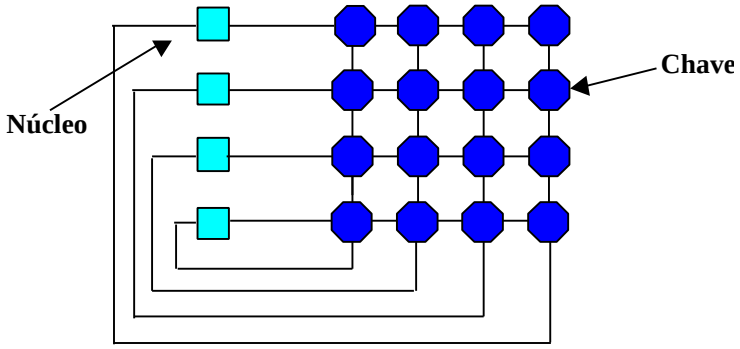


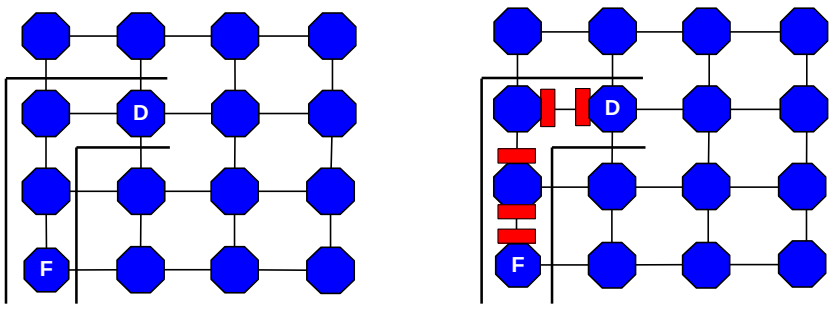
Figura 9 - Matriz de chaveamento 4 x 4.

Cabe ao projetista analisar qual dentre estas topologias adapta-se melhor a sua aplicação. Para isso o mesmo deve considerar fatores como: (*i*) relação custo/desempenho; (*ii*) números de núcleos que se pretende interligar à rede; (*iii*) grau da chave, ou seja, quantas portas de entrada e saída cada chave possui; (*iv*) escalabilidade, (*v*) confiabilidade, (*vi*) aplicação a que a rede se destina.

Para garantir a transferência de mensagem entre os núcleos, torna-se necessário impedir que situações como *deadlock*, *livelock* e *starvation* venham a ocorrer [DUA97]. *Deadlock* é definido como uma dependência cíclica entre as solicitações de acesso a recursos de comunicação e de armazenamento. *Livelock* ocorre quando pacotes ficam circulando pela rede sem se aproximar dos seus respectivos destinos. *Starvation* ocorre quando um pacote armazenado em um buffer solicita um canal de saída, sendo que este permanece bloqueado porque o canal de saída é sempre alocado para outro solicitante de mais alta prioridade.

A escolha da forma na qual pacotes são transferidos da entrada de uma chave para um de seus canais de saída poderá evitar os fenômenos descritos no parágrafo acima. Dois métodos de transferência de pacotes são utilizados: chaveamento de circuito e o chaveamento de pacotes.

- *Chaveamento de circuitos* (do inglês, *circuit switching*): inicialmente é estabelecido um caminho, denominado de *conexão*, do núcleo origem (fonte) até o núcleo destino, e logo após são enviados todos os pacotes. A Figura 10 ilustra esse método.

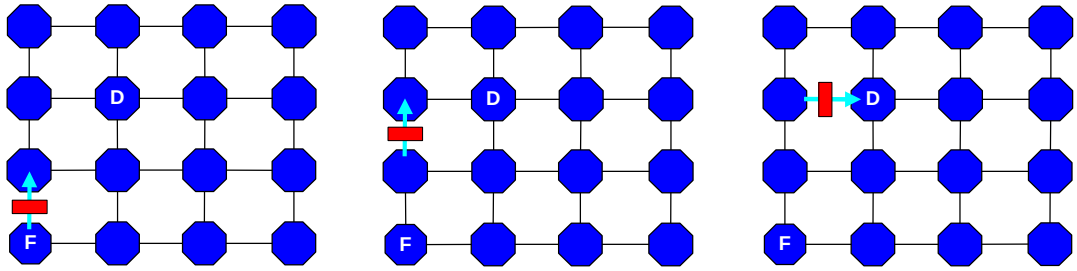


(a) Estabelecimento do caminho de comunicação **(b) Envio de pacotes do fonte (*f*) ao destino (*d*)**

As figuras acima (a, b) ilustram o modo de transferência de pacotes entre núcleos de uma rede, baseado no método de chaveamento de circuitos. Em um primeiro momento (figura a), é estabelecido o caminho de comunicação entre a chave fonte (F) e a chave destino (D), ou seja, determinam-se quais chaves receberão e enviarão pacotes até o seu devido destino. Após o estabelecimento do caminho de comunicação, são enviados os pacotes (figura b).

Figura 10 - Chaveamento de circuitos.

- *Chaveamento de pacotes* (*packet switching*): este método diferencia-se do chaveamento de circuitos pelo fato de que cada pacote informa a cada chave qual a direção que estes irão seguir na rede, ou seja, não existe um caminho pré-definido. A Figura 11 ilustra esse método.



(a) **(b)** **(c)**

As figuras acima (a, b, c) ilustram o modo de transferência de pacotes baseado no método de chaveamento de pacotes. Neste método o estabelecimento do caminho ocorre dinamicamente, ou seja, o pacote pode percorrer caminhos alternativos até chegar ao seu destino. Num primeiro momento (a), a chave fonte envia o pacote para seu vizinho, que a partir das informações sobre a direção que o pacote deve seguir, enviará o mesmo para a próxima chave (b) que fará o mesmo procedimento. Este procedimento ocorre até que o pacote chegue ao seu destino (c).

Figura 11 - Chaveamento de pacotes.

O chaveamento de circuitos ao estabelecer um caminho de comunicação para o envio de uma dada mensagem pode subtilizar os canais reservados (período sem envio de pacotes), desperdiçando largura de banda. Já no chaveamento de pacotes alocam-se apenas os canais necessários para transmissão dos pacotes, permitindo assim que mensagens diferentes compartilhem os demais canais da rede. Nesta abordagem, o custo para estabelecer o caminho de comunicação não existe, porém adiciona-se um custo de roteamento para cada chave visitada. A possibilidade de estabelecer caminhos dinamicamente permite que algoritmos de roteamento reajam mais rapidamente ao congestionamento[1] e a eventuais falhas na rede, optando por caminhos alternativos. O emprego do chaveamento de pacotes implica o uso de modos de chaveamento. Entre os utilizados citam-se *store-and-forward*, *virtual cut-through* e o *wormhole*, definidos a seguir.

- No modo *store-and-forward* (armazenar e passar) um pacote tem que ser completamente armazenado em um *buffer* antes de ser enviado para a próxima chave. Este modo gera latência na entrega dos pacotes e pode exigir grande armazenamento nas chaves, as quais devem ser dimensionadas para o tamanho máximo do pacote.

- No *virtual cut-through*, uma chave pode enviar um pacote a partir do momento que a próxima chave garanta poder receber todo o pacote. Sendo assim, torna-se necessário um *buffer* para armazenar o pacote completamente, se necessário, como no modo anterior. A vantagem deste modo em relação ao *store-and-forward* está na diminuição da latência de comunicação.

- O modo *wormhole* é uma variação do *virtual cut-through* com menor utilização de *buffers*. Neste modo, os pacotes são quebrados e transmitidos entre as chaves em unidades menores denominadas *flits*[2]. Uma desvantagem associada a este modo é que apenas o *flit* cabeçalho contém informações sobre o roteamento. Logo, os demais *flits* que compõem o pacote devem seguir o mesmo caminho reservado para o cabeçalho. Se um cabeçalho não puder avançar na

[1] Ocorre quando pacotes ficam esperando pela liberação de recursos que estão alocados.

[2] Unidade de controle de fluxo (do inglês, *flow control digit* ou *flow control unit*) [PEH00].

rede em função da contenção de recursos, todos os *flits* restantes são bloqueados ao longo do caminho, até que o caminho seja liberado.

Enquanto os modos de chaveamento definem como um pacote movimenta-se através das chaves, o algoritmo de roteamento define o caminho a ser utilizado por um pacote a partir do seu envio até o seu destino [NI93]. É importante salientar que o algoritmo de roteamento é dependente da topologia adotada.

Dependendo da onde as decisões são tomadas pode-se classificar o roteamento em *fonte* e *distribuído*. No roteamento *fonte* todo o caminho é decidido pela chave origem. Já no roteamento *distribuído* a rota do pacote é definida a cada chave visitada. Com isto, torna-se possível reduzir o tamanho do cabeçalho comparado com o roteamento fonte, que deve conter toda a rota no cabeçalho do pacote.

Em função do processo de seleção do caminho, o roteamento pode ser classificado como *determinístico* ou *adaptativo*. No roteamento *determinístico* o caminho é único, em função dos endereços das chaves origem e destino. O roteamento *adaptativo* prove vários caminhos possíveis da chave fonte para a chave destino. Nesta abordagem, o caminho de um pacote é estabelecido dependendo das condições da rede, como tráfego e congestionamento de canais. O roteamento adaptativo pode ser ainda classificado como *parcialmente* e *totalmente adaptativo*. No algoritmo *parcialmente adaptativo* utiliza-se um subconjunto dos caminhos disponíveis entre a origem e o destino, enquanto que no totalmente adaptativo é possível rotear um pacote através de todos os caminhos físicos.

O algoritmo de roteamento também pode ser *mínimo* ou *não-mínimo*. No primeiro algoritmo, o pacote deve aproximar-se do destino após cada chave visitada. No algoritmo não-mínimo, um pacote pode ser enviado para um caminho mais longo entre a chave origem e o destino.

2.2.2 Relação das camadas OSI-ISO no contexto de NoCs

Toda rede intra-chip possui um *protocolo de comunicação* que determina como um núcleo é conectado à rede, assim como, uma informação flui de sua fonte até seu destino [KUM03]. Assim como as redes de telecomunicação, o *protocolo de comunicação* em redes intra-chip é organizado em camadas, onde cada uma possui particularidades e funcionalidades de comunicação que apresentam níveis de hierarquia diferentes [BEN02][KUM02]. O modelo OSI é uma estrutura hierárquica composta por sete camadas que definem os requisitos para comunicação entre elementos de processamento [DAY83]. As redes intra-chip, geralmente, implementam um sub-conjunto das camadas inferiores. Como descrito em [GOO03], geralmente as camadas inferiores (física, enlace e rede) dependem da tecnologia de implementação. Por sua vez, as quatro camadas superiores são dependentes da aplicação. Estas camadas no contexto de redes intra-chip são descritas a seguir.

- *Camada Física*: Esta camada trata da organização física de interconexão da rede, ou seja, a distribuição dos canais de comunicação entre as chaves e os seus respectivos núcleos. Esta camada é responsável pelos detalhes referentes à transmissão e recepção de pacotes, sem a preocupação de detecção de erros em termos de hardware. Neste nível são definidos os

parâmetros elétricos dos sinais, a direção dos sinais e a largura dos canais.

- *Camada de Enlace de dados*: O principal propósito da camada de enlace é diminuir a falta de confiabilidade na transferência de dados sobre o meio físico. Esta camada define o protocolo para transmitir dados entre recursos da rede, como por exemplo, o protocolo *handshake*.

- *Camada de Rede*: A camada de rede determina a conexão entre um nodo origem e destino(s), estabelecendo o caminho que os pacotes irão utilizar. Neste nível o projetista deve analisar a sua aplicação a fim de escolher os algoritmos de chaveamento (*switching*) e roteamento (*routing*). Segundo Benini [BEN02], o roteamento e o chaveamento de redes intra-chip afetam o desempenho e o consumo de energia do sistema.

- *Camada de Transporte*: A camada de transporte é responsável pelo: (*i*) controle de fluxo; (*ii*) escolha do(s) algoritmo(s) de empacotamento e desempacotamento das mensagens e (*iii*) garantir a recepção ordenada dos pacotes. Uma diferença existente entre esta camada no contexto de SoCs para redes de comunicação é o tamanho do pacote, que logicamente é inferior.

- *Camada de Aplicação*: No contexto de SoC, as funcionalidades relevantes às três camadas superiores (aplicação, apresentação e sessão) do modelo OSI podem ser combinadas neste nível. Serviços importantes inerentes a essa camada incluem o gerenciamento e a sincronização de mensagens e conversão de formatos de dados pelo receptor.

Apresentadas as definições das camadas, é possível constatar que os serviços de comunicação dos três níveis inferiores do protocolo OSI devem ser implementados em cada chave da rede. Logo, os serviços correspondentes as camadas superiores são implementadas no próprio núcleo.

2.3 Estado-da-arte em redes intra-chip

Nesta Seção apresenta-se o estado-da-arte em redes intra-chip. Os resultados dessa revisão são apresentados resumidamente na Tabela 1. Esta Tabela[1] é o resultado de uma análise preliminar de publicações, onde cada linha corresponde a uma proposta de rede intra-chip cujos dados quantitativos ou qualitativos referentes a implementação encontrem-se disponíveis.

Os dados de implementação considerados relevantes estão divididos em três grupos: (*i*) dados da estrutura da chave, apresentados nas quatro primeiras colunas; (*ii*) dados de desempenho, nas três colunas seguintes; (*iii*) dados de implementação em silício e/ou prototipação, apresentados na última coluna.

Benini [BEN01][BEN02], De Micheli e Ye [YE03] contribuíram com artigos que apresentam os conceitos nesta área de pesquisa. Porém, nenhum desses documentos contêm detalhes referentes à implementação de redes intra-chip.

[1] Deve-se ressaltar, que tanto a Tabela como o texto apresentado nessa Seção, refletem o conteúdo da Seção 3 do artigo [MOR03a] aceito para publicação no periódico *Integration* (Elsevier).

Tabela 1 - Estado-da-arte em redes intra-chip.

Rede Intra-Chip	Topologia/ Chaveamento	Tamanho do *Flit*	Bufferização	Interface de Rede	Área da chave	Pico de desempenho	Suporte à QoS	Implement.
***SPIN* 2000**	Árvore gorda / determinístico e adaptativo	32 bits dados + 4 bits controle	Fila de entrada + 2 filas de saída compartilhadas	VCI	0.24 mm^2 CMOS 0.13µ	2 Gbits/s por chave		*Layout* ASIC 4.6 mm^2 CMOS 0.13µ
***ASOC* 2000**	Malha escalável / determinado pela aplicação	32 bits	None		50,000 transistores		Chaveamento de circuito (sem *wormhole*)	*Layout* ASIC CMOS 0.35µ
Dally 2001	*Torus* bidirecional 4x4 / XY – Fonte	256 bits dados + 38 bits controle	Fila de entrada		0.59 mm^2 CMOS 0.1µ (6.6 % of synchronous region)	4 Gbits/s por fio	Vazão garantida (canais virtuais)	Não
***NoCArc* 2001**	Malha (escalável)	290 bits dados + 10 bits controle	Fila de entrada ou saída					
Sgroi 2001	Malha	18 bits dados + 2 bits controle		OCP				
***Octagon* 2001**	Anel cordal - 8 nodes / distribuído e adaptativo	Tamanho de dados variável + 3 bits controle				40 Gbits/s	Chaveamento de circuito	Não
Marescaux 2002	*Torus* bidirecional (escalável) / XY bloqueante, hop-based, determinístico	16 bits de dados + 3 bits de controle	Fila de saída virtual	Proprietária	446 fatias da Virtex/Virtex-II (4.8% área *overhead*) for XCV800)	320Mbits/s por canal virtual a 40 MHz	2 canais virtuais multiplexidados por tempo	FPGA Virtex/ Virtex-II
Rijpkema 2002	Malha	32 bits	Fila de entrada		0.26 mm^2 CMOS 0.12µ	80Gbits/s por chave	Chaveamento de circuito (Vazão garantida)	*Layout* ASIC
***Eclipse* 2002**	Malha	68 bits	Fila de saída					Não
***Proteo* 2002**	Anel bidirecional	Tamanho de dado e controle variável		VCI				*Layout* ASIC CMOS 0.35µ
***HERMES* 2003**	Malha (escalável) / XY	8 bits dados + 2 bits de controle (parametrizável)	Fila de entrada (parametrizável)	OCP	631 LUTs 316 fatias da VirtexII	500 Mbits/s por chave a 25 MHz	Não	FPGA VirtexII
SoCIN 2003	Malha (escalável) / XY	n bits dados + 4 bits controle (parametrizável)	Fila de entrada (parametrizável)	VCI	420 LCs (n=8, sem buffer)	1G Mbits/s por chave a 25 MHz (n=8)	Não	Não

Uma escolha comum à maioria das redes intra-chip é o uso do *chaveamento de pacotes*, não estando esta característica comum exibida na Tabela 1. Uma exceção é a rede aSoC [LIA00], onde a definição das rotas das mensagens é fixada no momento da síntese do hardware. Dois conceitos associados, *topologia de rede* e *estratégia de chaveamento*, são os assuntos da primeira coluna da Tabela 1. A topologia de rede predominante na literatura é a malha. A razão para esta escolha deriva de três vantagens: (*i*) facilidade de implementação usando tecnologias planares de CI atuais; (*ii*) estratégia de chaveamento simplificada (XY); (*iii*) rede facilmente escalável. Outra topologia utilizada é a *torus* bidirecional, que pode ser utilizada para diminuir o diâmetro da rede [FOR02][MAR02]. O *torus* 2D dobrado [DAL01] é uma opção para redução do aumento do comprimento dos fios quando comparado a *torus* bidirecional. Um problema associado às topologias *torus* e malha é a latência de rede. Duas redes intra-chip utilizam topologias alternativas para obter redução de latência. A rede SPIN [GUE00][AND03][AND03a] e a chave proposta por [PAN03] adotam a topologia de árvore gorda, enquanto, a rede *Octagon* [KAR01][KAR02] sugere o uso da topologia anel cordal. Ambas as topologias conduzem a redes de menor diâmetro, com a conseqüente redução de latência. Observando as estratégias de chaveamento, é possível afirmar que existe uma carência de informações publicadas sobre algoritmos específicos. Isto indica a necessidade de pesquisas futuras nesta área. Exemplificando, é sabido que algoritmos de roteamento

adaptativos XY são propensos a *deadlock*, havendo soluções para evitar este fenômeno [GLA94]. Apesar disto, não há referência a estes algoritmos na literatura pesquisada.

O segundo parâmetro quantitativo importante das chaves é o *tamanho* do *flit*. A partir da Tabela 1 é possível classificar essa abordagem em dois grupos: (*i*) foco nas tecnologias dos futuros SoCs e (*ii*) redes intra-chip destinadas às tecnologias atuais. O primeiro grupo inclui as propostas de Dally [DAL01] e Kumar [KUM02], onde os canais de chaveamento possuem largura de 300 bits sem comprometer significativamente a área no SoC. Isso pode ser alcançado com o uso de tecnologias de 60 nm para implementar CI de dimensão 22 mm x 22 mm, com uma rede intra-chip 10 x 10, conectando 100 núcleos com dimensão 2 mm x 2 mm [KUM02]. Entretanto, isso ainda não é possível nos dias de hoje, ou seja, é uma proposta para futuros SoCs. O segundo grupo de trabalhos baseai-se em *flits* cujo tamanho varia entre 8 e 64 bits, uma largura de dado similar às arquiteturas de processadores atuais. Dois trabalhos apresentam redes intra-chip que foram prototipadas, Marescaux [MAR02] e a rede HERMES [MOR03]. Ambas as redes utilizam *flits* de menor tamanho, 16 bits (a rede HERMES possui largura parametrizável, tendo sido prototipada com largura de *flit* 8 e 16 bits).

O próximo parâmetro na Tabela 1 é a *estratégia de bufferização* da chave. A maioria das redes intra-chip adotam filas de entrada. Filas de entrada implicam numa fila simples por entrada, o que conduz a um menor aumento de área, quando comparado à chaves com filas nas saídas, o que justifica a escolha. Porém, filas de entrada apresentam o problema de bloqueio *head-of-line* (caso um *flit* fique bloqueado, todos os demais *flits* que entrarem na fila ficarão por conseqüência também bloqueados) [RIJ03a]. Uma forma de contornar este problema é através da utilização de filas de saída [FOR02], porém com um custo adicional de bufferização. Uma solução intermediária é a utilização de canais virtuais de saída associados com canais virtuais multiplexados por tempo, como proposto em [MAR02]. Outro parâmetro importante é o *tamanho da fila*, que implica na necessidade de solução em termos de compromisso entre a contenção da rede, latência de entrega dos pacotes e sobrecarga de área da chave. Filas grandes conduzem a pouca contenção de rede, alta latência de pacote e chaves com bastante área. Em contrapartida, filas pequenas implicam em situações opostas.

O último parâmetro estrutural consiste na *interface chave-núcleo* (interface de rede). Como descrito anteriormente, o uso de interfaces intra-chip padrão é uma tendência na indústria e no meio acadêmico, pois incrementa a reusabilidade dos núcleos. Uma rede intra-chip com interface de rede proprietária (protocolo próprio), como a proposta em [MAR02], pode diminuir a reusabilidade da rede devido à necessidade de se desenvolver invólucros para cada núcleo. O padrão de interface VCI é utilizado em três redes intra-chip: Proteo [SAA02][SAA02a][SAA03][SIG02]; SPIN [AND03][AND03a] e SoCIN [ZEF03][ZEF03a]. As redes propostas por Sgroi [SGR01] e Moraes [MOR03] adotam interfaces OCP.

A quinta coluna coleta os resultados referentes à área da chave. É interessante observar que as duas abordagens direcionadas a ASICs [DAL01][AND03a], ambas com *flits* de 32 bits, possuem área de 0.25 mm^2 para tecnologias similares. Os sistemas prototipados também apresentam resultados similares, em LUTs, de 631 a 890. Estes dados permitem ao projetista estimar qual o impacto que a rede intra-chip terá na área final e se este custo justifica a sua adoção.

A estimativa do pico de desempenho, apresentado na sexta coluna da Tabela 1, é um parâmetro que precisa de uma análise mais aprofundada para prover uma comparação significativa entre as diferentes redes intra-chip. Dessa forma, esta coluna mostra diferentes unidades para diferentes redes intra-chip. Esta coluna deve, conseqüentemente, ser considerada como uma ilustração de possíveis valores de desempenho. A maioria das estimativas são provenientes do produto de três valores: (*i*) número de portas da chave; (*ii*) tamanho do *flit* e (*iii*) freqüência de operação estimada. Nenhum dado de desempenho significativo foi encontrado nas publicações revisadas. A margem de variação dos valores deve-se, na maioria das vezes, aos dois últimos valores.

O próximo parâmetro da Tabela 1 diz respeito ao suporte de qualidade de serviço (QoS). A garantia de QoS mais comum entre as redes intra-chip analisadas é o chaveamento de circuito. Esta é uma forma de garantir a vazão e QoS para um determinado caminho de comunicação. Visto que muitas das propostas combinam chaveamento de circuito com técnicas de serviços garantidos, há o conseqüente aumento de área da chave. Este é o caso das redes intra-chip descritas em [DAL01][KAR01] e [RIJ03]. Canais virtuais são uma forma de garantir QoS sem comprometer a largura de banda, especialmente quando combinado com técnicas de multiplexação com divisão de tempo (TDM). Esta ultima técnica, exemplificada em [MAR02], evita que pacotes fiquem bloqueados por grandes períodos, desde que flits provenientes de diferentes entradas sejam alocados para uma dada fatia de tempo no canal virtual de saída. Espera-se que SoC atuais e futuros sejam dominados por aplicações do tipo multimídia. Conseqüentemente, o suporte de QoS é considerado como característica fundamental para redes intra-chip.

Finalmente, observa-se que resultados de implementação de redes intra-chip ainda estão escassos. Nenhuma da três implementações ASICs encontradas na literatura dão dicas se o projeto resultou em um circuito em silício. Três abordagens são apenas esboços de projeto. Em contrapartida, duas redes intra-chip relatadas foram prototipadas em FPGAs, [MAR02] e [MOR03].

2.4 Comparação entre as estruturas de interconexão de SoCs

Até o presente momento, neste Capítulo, foram apresentados os conceitos fundamentais sobre as estruturas de interconexão de SoCs, enfatizando redes intra-chip. A Tabela 2 apresenta uma comparação entre as estruturas de interconexão: (*i*) fios ponto-a-ponto dedicados; (*ii*) barramento; e (*iii*) redes intra-chip; frente aos seguintes critérios:

- *paralelismo*, as estruturas (*i*) (*iii*) permitem que múltiplas comunicações sejam realizadas simultaneamente. Já a estrutura de interconexão (*ii*), permite apenas uma comunicação por vez, dado a concorrência dos núcleos pelo barramento. A utilização de barramentos interconectados através de pontes pode prover paralelismo, limitado ao número de barramentos independentes que compõem a arquitetura.

- *consumo de energia*, as estruturas (*i*) (*iii*) tentem a utilizar fios mais curtos o que resulta em menor consumo de energia em função da menor carga capacitiva. Em contrapartida, a estrutura (*ii*) tende a utilizar fios mais longos o que aumenta o consumo de energia. Além disso, existe um aumento de carga capacitiva a cada núcleo acrescentado ao barramento, o que vem a reduzir

o desempenho do sistema. No caso de redes intra-chip não acrescenta-se capacitâncias, pois os núcleos são conectados ponto-a-ponto à porta local da chave.

- *escalabilidade:* na estrutura (*i*) para cada núcleo adicionado ao sistema existe a necessidade de implementação de novos canais para a comunicação. Para estruturas do tipo (*ii*) a escalabilidade é limitada a dezenas de núcleos [ZEF03a]. A estrutura (*iii*) é escalável, pois o acréscimo de chaves à rede tende a aumentar o desempenho total do sistema, pois o número de canais de comunicação cresce.

- *reusabilidade:* As estruturas (*ii*) (*iii*) são consideradas reutilizáveis pois estas podem ser utilizadas em projetos distintos. Por sua vez, a estrutura (*i*) é projetada para uma situação especifica o que reduz o seu reuso em outros projetos.

Tabela 2 - Comparação entre as estruturas de interconexão mais utilizadas em SoCs e a abordagem baseada em redes intra-chip.

Estruturas de Interconexão	Fios Ponto-a-ponto dedicados	Barramento	Redes intra-chip
Paralelismo	As conexões ponto-a-ponto são independentes, logo, é possível haver paralelismo.	Permite apenas uma comunicação por vez. Barramentos hierárquicos podem prover paralelismo, limitado ao número de barramentos independentes.	Comunicação também ponto-a-ponto, com paralelismo suportado.
Consumo de Energia por fio	Menor, se comparado à barramento. Motivo: fios mais curtos	Maior. Motivo: fios mais longos, além de aumento de carga capacitiva a cada núcleo acrescentado.	Menor, se comparado à barramento. Fios mais curtos. Novos núcleos não acrescentam capacitâncias, à estrutura de interconexão.
Escalabilidade	Não oferece escalabilidade. Motivo: projetado para uma situação especifica.	Limitada a dezenas de núcleos.	Escalável através do acréscimo de mais chaves à rede.
Reusabilidade	Reuso bastante restrito. Motivo: projetado para uma situação específica.	Totalmente reutilizáveis.	Totalmente reutilizáveis.

2.5 Considerações sobre integração de núcleos a redes intra-chip

Um aspecto importante na abordagem de redes intra-chip consiste em como integrar núcleos à rede, garantindo a comunicação entre estes através do meio de comunicação. A menos que o núcleo atenda ao *protocolo de comunicação* da rede, torna-se necessário criar um *invólucro* que permita integrá-lo à mesma. Um invólucro deve possibilitar a integração física (interface – largura de sinais, sinais de entrada e saída) e os serviços de comunicação[1] (segmentação e remontagem dos pacotes) entre o núcleo e a rede. No contexto de redes intra-chip denomina-se esse invólucro de *interface de rede* (IR).

Segundo Kumar [KUM03], é possível dividir o projeto interno da IR em duas partes: (*i*) a parte específica à rede (independente do núcleo), responsável pela temporização, bufferização e aspectos de sincronização durante a transmissão/recepção de dados; (*ii*) parte específica ao núcleo,

[1] Utiliza-se nesse trabalho o termo protocolo de rede como sinônimo para serviços de comunicação.

responsável pela montagem/desmontagem dos pacotes. A Figura 12 ilustra um núcleo integrado à rede através da IR núcleo-chave.

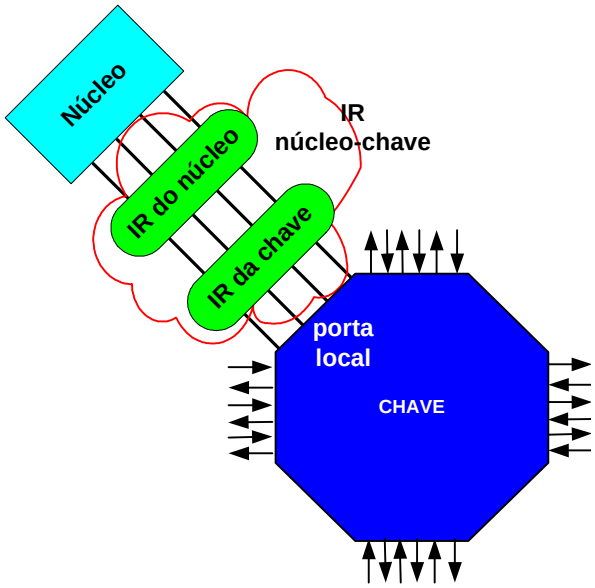


Figura 12 - Interface de conexão entre o núcleo e a porta local da chave.

Praven [PRA03] discute a implementação da interface de rede em dois aspectos: (*i*) como implementá-la, em software ou em hardware; e (*ii*) onde implementá-la, ou seja, do lado do núcleo ou do lado da chave. A diferença básica consiste na simplicidade ou na dificuldade de implementar a IR, e quanto de trabalho se coloca em cada lado. Além de considerar os aspectos (*i*) (*ii*), a divisão de esforços no projeto pode aumentar/diminuir a reusabilidade da IR (principalmente do lado da rede) e a proteção da propriedade intelectual de ambos os lados. Dentro desse contexto, apresentam-se duas abordagens:

- *o projetista do SoC desenvolve os dois lados da interface de rede* (núcleo e chave). Nesta abordagem, o projetista (integrador do sistema) deve conhecer tanto a operação do(s) núcleo(s) como a da chave, para desenvolver a interface de rede de cada núcleo e da chave. Este fato além de reduzir a proteção da propriedade intelectual e de aumentar o tempo de projeto, pode comprometer a reusabilidade da interface de rede, dado a diversidade de núcleos que podem compor um SoC.

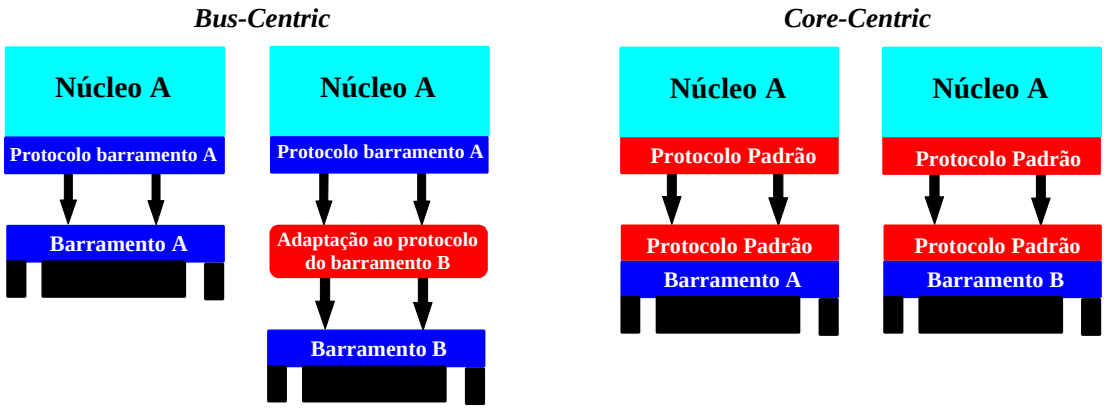
- *o(s) projetista(s) do(s) núcleo(s) e o projetista da chave desenvolvem apenas o seu lado da IR, utilizando-se um protocolo comum para a comunicação entre as partes*. Logo, o meio comum entre os núcleos e a chave (rede intra-chip) é o protocolo de comunicação adotado, que *preferencialmente* deve ser padrão. O Capítulo seguinte apresenta exemplos de protocolos de comunicação padrão. Esta abordagem pode resultar no aumento do reuso dos núcleos e da chave, e no acréscimo da proteção intelectual às entidades envolvidas, pois não existe a necessidade de "transmissão" de conhecimento referente à operação do(s) núcleo(s) ou da chave. Além das informações inerentes ao protocolo padrão (e.g. tipos de transações possíveis), os projetistas de núcleos precisam apenas conhecer o protocolo de comunicação da rede (camada de rede), da mesma forma o projetista da rede intra-chip pode projetá-la sem conhecimento de quais núcleos serão conectadas à mesma.

Deve-se ressaltar que independentemente da abordagem utilizada, a segmentação e a remontagem dos pacotes (criação dos pacotes, definição da estratégia de endereçamento de núcleos, inserção de campos para correção de erro, bufferização dos dados) pode ser realizada tanto no *lado da rede* como *no lado núcleo*. Entretanto, recomenda-se que a maior parte da segmentação e remontagem dos pacotes ocorra *no lado núcleo*, pois isto corresponde às camadas superiores (transporte e aplicação). Acredita-se que as IRs da rede intra-chip devam ser idênticas, o que pode favorecer a reusabilidade da mesma, conforme já mencionado ao final da Seção 2.2.2.

3 INTERFACES PADRÃO PARA INTERCONEXÃO DE NÚCLEOS

A indústria de semicondutores vem concentrando esforços no que diz respeito ao incremento do reuso de núcleos, conforme apresentado na introdução deste trabalho. A reusabilidade de núcleos ainda está longe do desejado, mas padronizar as interfaces desses núcleos pode conduzir ao aumento de reusabilidade, facilitando a interconexão desses em diferentes SoCs. Entre as interfaces de comunicação padrão mais difundias na comunidade de semicondutores cita-se: VCI (*Virtual Component Interface*) da VSIA (*Virtual Socket Iniciative Alliance*) e OCP (*Open Core Protocol*) da *Sonics* [OCP02b].

O uso de interfaces padrão permite a migração de uma abordagem *bus-centric* [OCP02c] para uma abordagem *core-centric*. Na abordagem *core-centric* o projetista concentra-se no projeto do núcleo e não do sistema como um todo.



As figuras acima ilustram as abordagens bus-centric e core-centric. Na abordagem bus-centric, um núcleo pode ser conectado diretamente ao barramento alvo, dado a similaridade do protocolo de comunicação A. Porém, para conectar o mesmo núcleo a um barramento B (protocolo diferente do A) torna-se necessário adaptar o núcleo ao protocolo B. Na abordagem core-centric, o núcleo A, dotado de um protocolo padrão, pode ser conectado ao barramento A e B (caso ambos possuam interfaces padrão similares) sem a necessidade de adaptação de protocolo. Ou seja, alterações em termos de sinais e métodos de comunicação são desnecessárias.

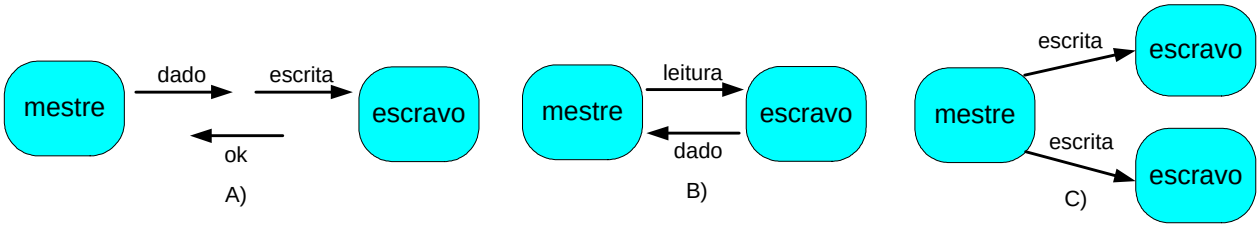
Figura 13 - Abordagem *bus-centric* versus *core-centric*.

3.1 Padrão de Comunicação VCI

VCI é um padrão de interface que permite interconectar núcleos a um SoC. Um núcleo VCI deve ter uma interface compatível com o padrão, caso contrário, o mesmo deve ser envolvido por uma lógica de adaptação de protocolo. Neste padrão, os núcleos de um sistema são conectados ponto-a-ponto, sendo a comunicação entre interfaces mestre-VCI e escravo-VCI realizada através de transações, que consistem de requisições e possíveis respostas a estas.

Uma transação é composta basicamente por: (*i*) um comando que descreve uma ação que deve ser executada pelo lado escravo; (*ii*) dado de saída enviado pelo lado mestre junto com um comando que necessita de um dado para ser executado (e.g. execução de uma escrita); (*iii*) dado de resposta - conseqüência de uma transação executada pelo escravo que resulta em dados; (*iv*) aceite/finalização da transação – ativando ou negando um ou mais sinais quando uma transação é

aceita ou finalizada. Exemplos de transações são ilustrados na Figura 14.



A figura acima ilustra três tipos de transações: (a) transação de escrita com aceite de operação; (b) transação de leitura simples que retorna dados, sem a necessidade de aceite do lado escravo; (c) multi-cast – um mestre inicia transações para múltiplos destinos.

Figura 14 - Exemplos de transações.

É possível classificar interfaces VCI em três tipos: (*i*) mestre; (*ii*) escravo; e (*iii*) mestre-escravo. Uma interface do tipo *mestre* é aquela que tem a capacidade de começar uma transação através de uma requisição para um determinado escravo. Já uma interface do tipo escravo deve apenas responder a requisições enviando uma resposta ao núcleo solicitante. Um núcleo que possua uma interface do tipo mestre-escravo (por exemplo, um co-processador) possui ambas funcionalidades, ou seja, este pode tanto solicitar como atender a uma requisição.

Cada núcleo deve implementar a interface conforme a sua funcionalidade. Por exemplo, um processador deve implementar uma interface do tipo mestre-VCI, enquanto que uma memória necessita de uma interface do tipo escravo-VCI. Já um barramento necessita dos três tipos de interfaces, dada a possibilidade de interconexão de núcleos que possuam alguma das três interfaces descritas acima. Cada interface pode ser baseada em uma das três classes de sinais suportadas pelo padrão, sendo estas [BIR99]: (*i*) VCI básico; (*ii*) VCI periférico; (*iii*) VCI avançado. A documentação do protocolo em questão é exclusiva para membros associados. Tal fato impossibilitou uma descrição mais detalhada do mesmo.

3.2 Padrão de Comunicação OCP

Da mesma forma que VCI, o padrão de comunicação OCP define uma interface ponto-a-ponto entre dois núcleos. Neste contexto, um núcleo deve atuar como mestre e o outro como escravo. Apenas o mestre (iniciador) pode enviar comandos inicializando as transações. O escravo por sua vez responde aos comandos informados, tanto recebendo como passando dados ao mestre. Basicamente, existem dois tipos de comandos: *escrita* e *leitura* de dados (palavras). A Figura 15 ilustra dois tipos de transações geradas a partir dos comandos de escrita e leitura.

Extensões para esses comandos são: no caso de escrita, *broadcast*, e de leitura, *ReadEx*. O comando *broadcast* diferencia-se do comando de escrita por permitir a transferência de dados de um núcleo mestre-OCP para vários ou todos os núcleos conectados à interface escravo-OCP. Por exemplo, várias memórias conectadas a um barramento (ou uma chave) com interface escravo-OCP. O segundo comando de extensão *ReadEx*, leitura exclusiva (do inglês, *read exclusive*), garante acesso exclusivo a um determinado endereço de memória. Três comandos foram adicionados na versão 2.0 [OCP03]. São estes: *ReadLinked*, *WriteNonPost* e *WriteConditional*.

A partir dos comandos descritos acima, torna-se possível transferir palavras entre interfaces OCP. O número máximo de bits (*word size*) de uma palavra que pode ser transferida em

uma operação OCP simples limita-se a: 8, 16, 32, 64 e 128 bits. Deve-se ressaltar que OCP baseia-se no conceito de extensão de zeros (*zero-extended*), ou seja, utilizam-se zeros para preencher uma palavra com largura menor que as permitidas pelo protocolo. Esta restrição não está presente na versão 2.0 do protocolo [OCP03]. Dessa forma, o usuário poderá configurar a largura da palavra sem um conjunto discreto de valores específicos.

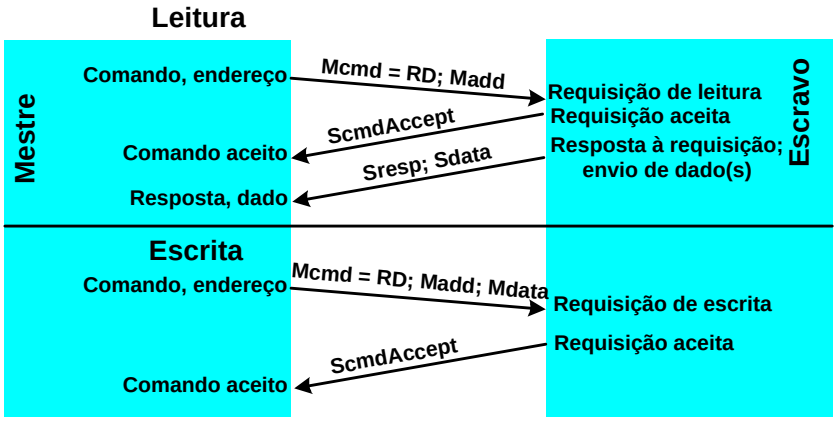


Figura 15 - Exemplo de transações de escrita e leitura OCP.

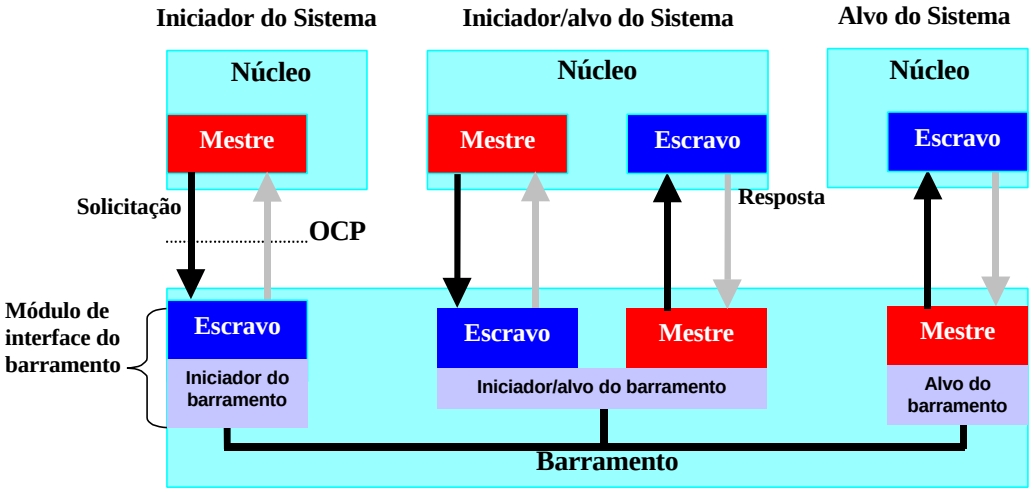
As transações de escrita e leitura de palavras podem acontecer separadamente, ou seja, respostas para requisições podem ocorrer em ciclos diferentes. Além de transações simples (escrita e leitura), OCP possui suporte para transações em modo rajada (do inglês, *burst*). O modo rajada caracteriza-se pela transmissão de dados em volume, requerendo acréscimo de sinais na interface OCP. A utilização desse modo pode ser necessária para núcleos que atuem com alta transferência de dados, como um núcleo DSP.

Além do modo de transmissão baseado em rajada, OCP suporta requisições e respostas fora de ordem. Isto é possível a partir da utilização de identificadores de requisição e respostas, denominados *threads*. Neste contexto, quando um núcleo com interface mestre-OCP efetua uma requisição, junto é enviado o identificador da *thread* (interna a sua interface) correspondente a essa requisição. Da mesma forma, com toda resposta é provido o identificador referente a mesma. Logo, o mestre fica sabendo qual requisição foi efetuada com sucesso.

A Figura 16 ilustra um sistema simples contendo um barramento empacotado (com interfaces OCP para o mundo externo) e três núcleos: um atuando apenas como mestre, outro apenas como escravo e um terceiro podendo ser tanto mestre como escravo.

3.2.1 Sinais OCP

Uma interface OCP é composta por um conjunto de sinais que podem ser utilizados para dar suporte às particularidades de um núcleo. Ou seja, as particularidades inerentes ao núcleo indicam quais sinais devem compor a sua interface e o invólucro OCP no qual este será inserido. Todos os sinais OCP são ponto-a-ponto, unidirecionais e são amostrados na borda de subida do relógio. Estes sinais são classificados em três grupos: (*i*) sinais de fluxo de dados, (*ii*) sinais opcionais de controle, (*iii*) e sinais de teste.



A figura descreve o modelo de comunicação OCP. Observar que este é sempre ponto-a-ponto, ou seja, torna-se necessária a comunicação entre pares mestre-escravo. Notar também que o módulo OCP escravo interno ao barramento, atua de fato como mestre do barramento.

Figura 16 - Sistema mostrando um barramento e núcleos empacotados com instâncias OCP.

Sinais de fluxo de dados: correspondem a um conjunto de sinais que podem ser utilizados para garantir o fluxo de dados entre dois módulos. Divide-se este conjunto em: *sinais básicos* e *sinais opcionais* (simples e complexos). Os *sinais adicionais* podem ser utilizados para atender particularidades de um dado módulo, e.g., permitir transações em modo rajada.

Sinais opcionais de controle: relaciona-se esse grupo a transmissão de informações de controle como *reset* e interrupções.

Sinais de teste: referem-se a um grupo de sinais opcionais utilizados para testar um determinado núcleo. A utilização desses sinais suporta o padrão IEEE 1149.1 [IEE02a].

A direção dos sinais descritos acima depende do modo de atuação do núcleo, sendo que este pode atuar como: mestre/escravo e/ou sistema/núcleo. Um núcleo que atua como “sistema” é responsável por gerar sinais de controle. Por outro lado, um núcleo que atua como “núcleo” não gera esses, utiliza-os. A Tabela 3 ilustra as seis combinações possíveis.

Tabela 3 – Tipos de núcleos OCP.

		Com sinais de controle	
	Sem sinais de controle	**Atuando como sistema**	**Atuando como núcleo**
Núcleo atuando como Escravo OCP	Escravo	Sistema Escravo	Núcleo Escravo
Núcleo atuando como Mestre OCP	Mestre	Sistema Mestre	Núcleo Mestre

Dentre todos os sinais suportados pelo protocolo destaca-se o grupo de sinais básicos. Justifica-se essa afirmativa porque estes sinais devem compor a interface de qualquer núcleo dotado de interface OCP. Estes sinais são descritos na Tabela 4.

Tabela 4 - Conjunto de sinais básicos OCP.

Nome	Largura	Parâmetro de configuração de largura	Controle	Função do sinal
Clk	1	Fixo	Variável	Relógio OCP
Maddr	1-32	Addr_wdth	Mestre	Endereço de Transferência
MCmd	3	Fixo	Mestre	Endereço de comando
Mdata	8/16/32/64/128	Data_wdth	Mestre	Escrita de Dados
SCmdAccept	1	Fixo	Escravo	Aceitação de transferência
Sdata	8/16/32/64/128	Data_wdth	Escravo	Leitura de Dados
Sresp	2	Fixo	Escravo	Resposta de Transferência

Na versão 2.0 deste protocolo, o conjunto de sinais básicos foi reduzido [OCP03]. Por exemplo, os sinais *Mdata* e *Sdata* não precisam compor a interface de um núcleo OCP. Maiores informações referentes aos sinais OCP podem ser obtidas na especificação do protocolo [OCP02][OCP02a].

3.2.2 Compatibilidade entre núcleos OCP

A utilização do protocolo padrão em questão *não garante a comunicação* entre todo e qualquer núcleo com interface OCP. Ou seja, é possível definir várias interfaces OCP todas *incompatíveis* entre si. Uma comunicação OCP é garantida à medida que as interfaces dos núcleos são *compatíveis*. Sendo assim, as "similaridades" das interfaces devem permitir a conexão e a comunicação entre dois núcleos que adotam OCP. Neste contexto, duas interfaces são ditas *compatíveis* se estas atendem à três quesitos: (*i*) compatibilidade de núcleos; (*ii*) compatibilidade de protocolo; (*iii*) compatibilidade de sinais.

- *Compatibilidade de núcleos*: diz respeito ao modo de atuação dos núcleos. Sendo assim, um núcleo com interface mestre-OCP pode comunicar-se somente com outro que possua uma interface escravo-OCP (ou mestre-escravo). Da mesma forma, um núcleo que apresenta sinais de sistema em sua interface deve comunicar-se com outro que possua uma interface do tipo núcleo.

- *Compatibilidade de protocolo*: refere-se ao suporte de comandos apresentados entre as interfaces. Por exemplo, se a interface de um núcleo suporta o comando de leitura exclusiva a outra interface também deve suportar este comando.

- *Compatibilidade de sinais*: reporta-se à necessidade de equivalência dos sinais utilizados, e da largura destes [OCP02].

3.2.3 Temporização e Modos de Transferência

Esta Seção apresenta alguns dos modos de transferência que podem ser implementados com o conjunto de sinais básicos do protocolo em questão. Objetiva-se com isso, apresentar ao leitor o comportamento do protocolo OCP. Outros diagramas de transferência podem ser

encontrados na especificação OCP [OCP02].

3.2.3.1 Diagrama de uma transferência simples de escrita e leitura de dados.

A Figura 17 ilustra o diagrama de tempo para uma transferência simples de escrita e leitura.

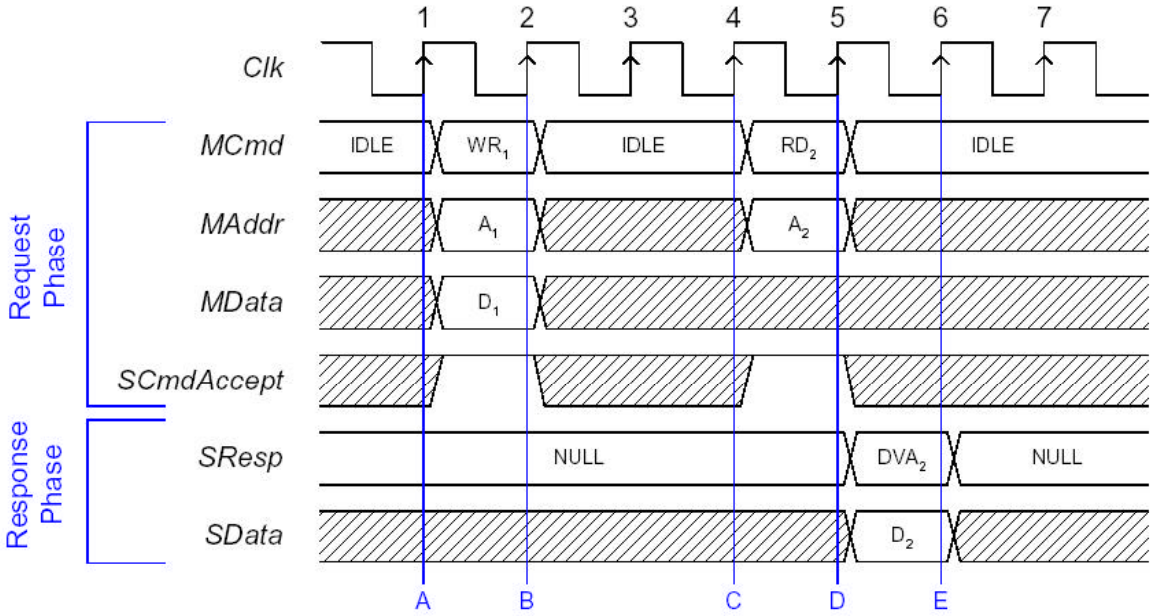


Figura 17 - Diagrama de tempos de uma escrita e leitura simples.

A seqüência para transferência segue os seguintes eventos:

A: O mestre-OCP inicia a fase de requisição quando ocorre a transição de IDLE para WR no sinal *Mcmd*. No mesmo instante é apresentado um endereço de memória válido (A_1) no sinal *Maddr*, e um dado válido no sinal *Mdata*. Estes três sinais devem ser apresentados juntos, conforme a semântica do protocolo. O escravo-OCP por sua vez, ativa o sinal de *ScmdAccept* no mesmo ciclo, permitindo a transferência (transação de escrita) sem latência.

B: Nesse instante, o escravo-OCP captura os valores referentes ao endereço, aos dados e utiliza estes internamente para efetuar a escrita. Como o *ScmdAccept* está ativo, isto indica término da fase de requisição.

C: O mestre-OCP inicia a fase de requisição de leitura (RD_2) atribuindo o comando de leitura RD, no sinal *MCmd*. Neste momento o mestre apresenta um endereço de memória válido (A_2) no sinal *Maddr*.

D: O escravo-OCP captura o endereço do sinal *Maddr* e utiliza este internamente para determinar o dado a ser retornado. A fase de resposta começa quando o comando *SResp* passa de NULL para DVA. No mesmo instante o escravo apresenta o dado no sinal *SData*. Como o *ScmdAccept* está ativo, isto indica término da fase de requisição.

E: Ao perceber que o campo *SResp* indica um dado válido (DVA) o mestre-OCP captura o dado do sinal *SData*, finalizando a fase de resposta. A latência gerada por essa fase é igual a 1 ciclo de relógio.

3.2.3.2 Requisição com protocolo *Handshake*

A Figura 18 ilustra um diagrama da requisição de escritas do tipo *handshake*. Para cada uma das três requisições de escrita o sinal *ScmdAccept* é ativo, informando a aceitação da transação. Este diagrama caracteriza aceitações com latências diferentes.

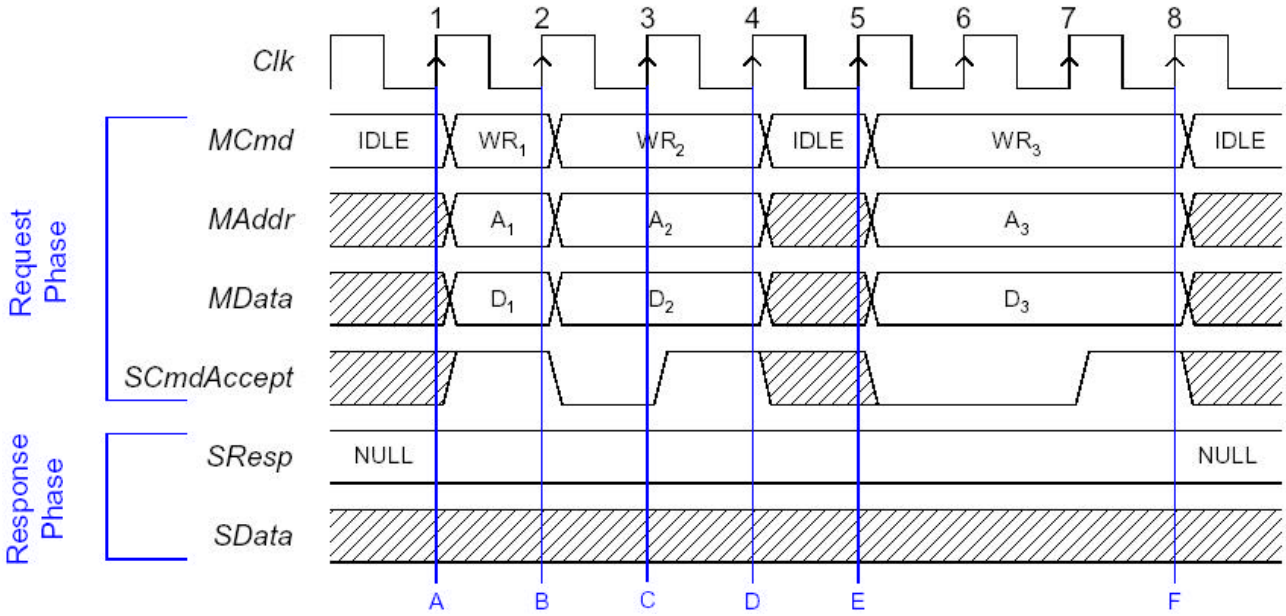


Figura 18 - Diagrama de tempo para um requisição do tipo *handsshake*.

A seqüência para transferência segue os seguintes eventos:

A: O mestre-OCP inicia a fase de requisição apresentando o comando de WR_1 no sinal *MCmd*. No mesmo instante é apresentado um endereço válido (A_1) no sinal *Maddr* e um dado no sinal *MData*. O escravo por sua vez, ativa o sinal de *ScmdAccept* no mesmo ciclo, gerando assim uma resposta para requisição sem latência (como apresentado anteriormente).

B: O mestre-OCP inicia, no próximo ciclo de relógio, uma nova fase de requisição. O escravo-OCP, por sua vez, captura o endereço de escrita e o dado. Neste instante, o *ScmdAccept* está desativado, o que indica que o escravo-OCP não está pronto para atender a nova requisição de escrita.

C: O mestre-OCP mantém os valores nos sinais (*MCmd, MAddr, MData*) até o escravo ativar o sinal *ScmdAccept.* Com um ciclo de relógio de latência o escravo ativa o sinal *ScmdAccept.*

D: O escravo captura o endereço de escrita e o dado.

E: Depois de permanecer um ciclo de relógio em IDLE, o mestre começa uma nova fase de requisição de escrita.

F: O *ScmdAccept* é ativado com latência de dois ciclos de relógio. Neste instante, o escravo captura os valores dos sinais *MCmd, MAddr, MData*.

3.2.3.3 Requisição com *Handshake* e Resposta Separada

A Figura 19 ilustra uma requisição de leitura com latência de aceitação e resposta. A aceitação para requisição de leitura apresenta uma latência igual a 2, que corresponde ao número de ciclos de relógio que o *ScmdAccept* permaneceu desativado. Entre a fase e requisição e a fase de resposta apresenta-se uma latência de três ciclos de relógio, referentes ao número de ciclos de relógio do final da fase de requisição (D) para o final da fase de resposta (F).

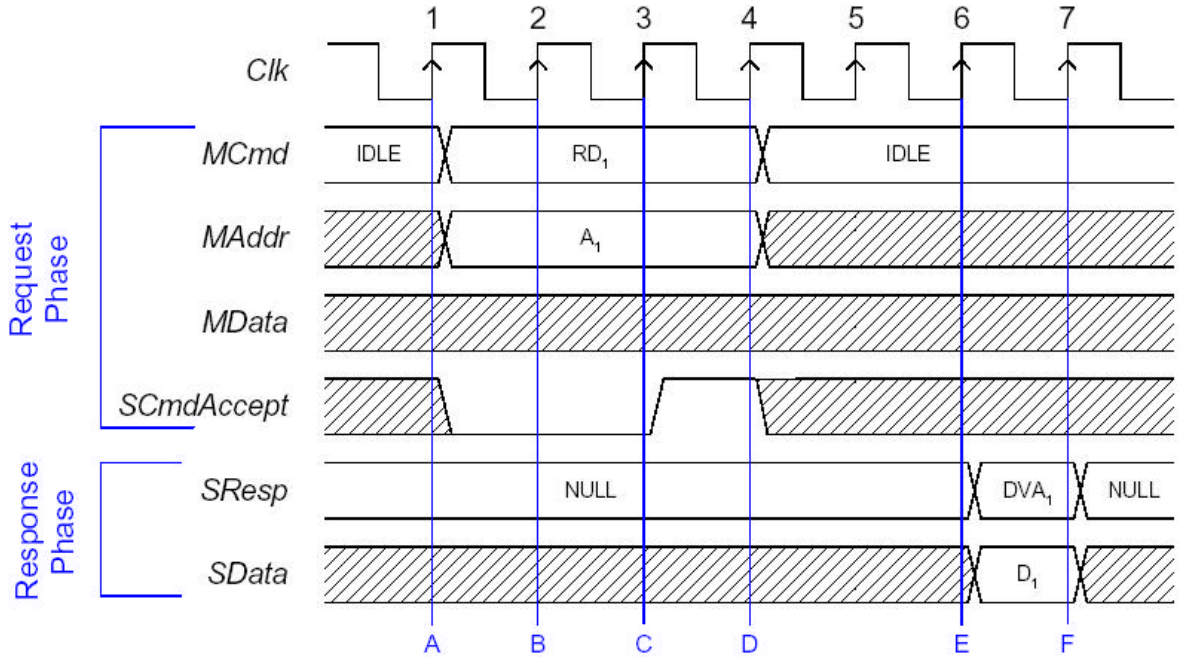


Figura 19 - Diagrama de tempo para requisição de *handshake* e resposta separada.

A seqüência para transferência segue os seguintes eventos:

A: O mestre-OCP inicia a fase de requisição de leitura apresentando o comando de RD_1 no sinal *MCmd*. No mesmo instante apresenta-se um endereço válido (A_1) no sinal *Maddr*. O escravo por sua vez, não está pronto, sendo assim o *ScmdAccept* permanece desativado.

B: O mestre-OCP vê que o *ScmdAccept* não está ativo, e mantêm todos os valores dos sinais da fase de requisição.

C: O escravo-OCP ativa o sinal de *ScmdAccept.* O mestre-OCP continua mantendo os sinais na fase de requisição. O campo *Sresp* continua como NULL.

D: O escravo-OCP captura o endereço (A_1). Porém, este não está pronto para responder, sendo assim, o sinal *SResp* continua em NULL.

E: O escravo-OCP apresenta uma resposta válida, ou seja, o sinal de *SResp* recebe DVA. No mesmo momento o dado é colocado no *SData*.

F: O mestre-OCP recebe a resposta e captura o dado de leitura.

3.2.4 Certificação de Núcleos OCP

A certificação de um núcleo frente a um padrão de comunicação é de fundamental importância tanto para projetista como para o usuário. De um lado, o projetista precisa garantir que seu produto (núcleo) atenda às normas do protocolo padrão adotado. O usuário, por sua vez, quer utilizar o núcleo no seu projeto, sem se preocupar com a validação do mesmo. Caso contrário, a possível reusabilidade conquistada com a padronização de comunicação é perdida frente a necessidade de verificação do padrão. A ferramenta *CoreCreator* [SON03] permite ao projetista verificar se um núcleo atende à especificação do protocolo OCP. Deve-se salientar, que a ferramenta *CoreCreator* suporta apenas núcleos descritos em VHDL e Verilog. As etapas de certificação de núcleos OCP são descritas abaixo e ilustradas na Figura 20.

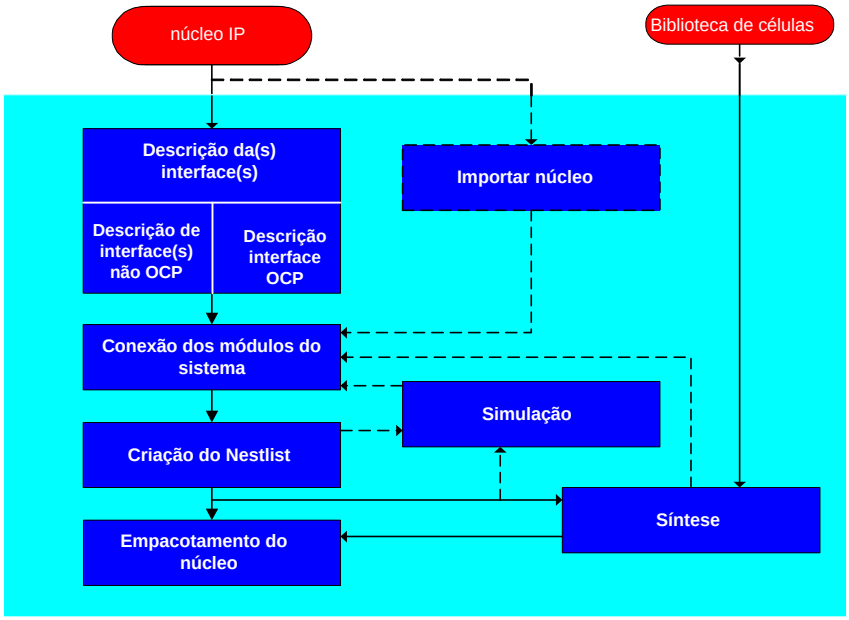


Figura 20 - Fluxo de certificação da ferramenta *CoreCreator*.

Para poder importar um núcleo na ferramenta *CoreCreator* é necessário descrever a(s) interface(s) mais externa(s) em um formato que a mesma possa interpretar. Dentro desse contexto, o usuário pode proceder de duas formas:

(i) *descrever as interfaces do núcleo e do invólucro OCP:* Este caso é indicado quando o invólucro OCP-núcleo e o núcleo propriamente dito correspondem a descrições e interfaces separadas.

(ii) *descrever apenas a interface OCP:* Neste caso, o módulo utilizado no *CoreCreator* corresponde ao núcleo completo, invólucro e o núcleo propriamente dito.

A primeira parte do fluxo de certificação, denominada *descrição das interfaces*, conforme a Figura 20, corresponde a duas etapas:

- *Descrição de interfaces não-OCP*: geração manual do arquivo que reflete a interface não-OCP do núcleo, caso o projetista opte pela opção (*i*) acima. Estas interfaces são definidas como *bundles* (conjunto de sinais). A Figura 21 ilustra a descrição da interface de um núcleo (RAM) que é descrito na Seção 3.2.5 deste trabalho.

```
bundle "ram" {        definição do bundle
    #"version 1.0"
    interface_types memory controller

    net ce_n { direction input output
               width 1}
    net we_n { direction input output
               width 1}
    net oe_n { direction input output
               width 1  }
    net address { direction input output
                  width 19}
    net data { direction inout inout
               width 16}
# end the bundle ram
```

sinais da interface do núcleo (RAM)

Figura 21 - Exemplo da descrição da interface não-OCP de um núcleo.

- *Descrição da interface OCP:* nesta etapa deve-se criar um arquivo que represente as características da interface OCP do núcleo a ser validado. Neste exemplo, especifica-se a interface OCP ("*slave_ocp_ram*") do tipo escravo (parâmetro *interface_type slave)* do invólucro que contém o núcleo (e.g. RAM) e a lógica OCP. As características OCP são definidas a partir dos parâmetros de configuração dos sinais, conforme ilustrado na Figura 22.

```
interface "slave_ocp_ram" bundle ocp { # especificação da interface OCP

interface_type slave   #definição do tipo de interface OCP
  # no prefix
    prefix ""
    # parâmetros para definição dos sinais OCP
    param reset 1
    param addr_wdth 19
    param data_wdth 16
    param readex_enable 0
    param broadcast_enable 0 # parâmetro não utilizado
}
```

Figura 22 - Exemplo da descrição de uma interface OCP de um núcleo.

Estas duas descrições serão associadas ao núcleo que iremos validar na ferramenta C*oreCretor*. A etapa seguinte, denominada "*Conexão dos módulos do sistema*" é responsável por conectar:

- *módulo que teve a interface criada na etapa anterior*: invólucro contendo o núcleo com interface OCP a ser validado (*W_ram* - índice 1 da Figura 23).

- *monitor OCP*: módulo utilizado para verificar as transações OCP (índice 2 da Figura 23).

- *módulos de comportamento* (índice 3 da Figura 23): utilizados para modelar o comportamento de núcleos ou sistemas, que devem gerar (mestre) ou responder (escravo) à estímulos OCP. Estes módulos são divididos em dois grupos: (*i*) *QS-Models* e (*i*) *QC-Models*. Os módulos (*i*) servem para modelar um sistema conectado ao núcleo do usuário. Por sua vez, os módulos (*ii*) são utilizados para modelar núcleos que atuem como mestre (*QC-Master*) ou escravo (*QC-Slave*). O comportamento dos módulos *Q-Master* podem ser definidos a partir da linguagem STL. STL é uma linguagem de descrição de tráfego e transações OCP utilizada para exercitar a interface de um núcleo com interface escravo-OCP. Os módulos *Q-Slave* podem atuar como memórias ou filas (FIFO) para leitura/escrita de dados. É possível definir a latência de reposta para as requisições provindas de um núcleo com interface mestre-OCP.

- *módulos que não possuem interface OCP* (se for o caso, não ilustrado na Figura 23).

A Figura 23 ilustra a interface da ferramenta *CoreCreator* com os módulos acima conectados entre si. Além desses módulos, a Figura 23 apresenta algumas das fases de certificação. As principais fases são: (*i*) criação do *netlist*; (*ii*) preparação da simulação; (*iii*) simulação; e (*iv*) análise de resultados.

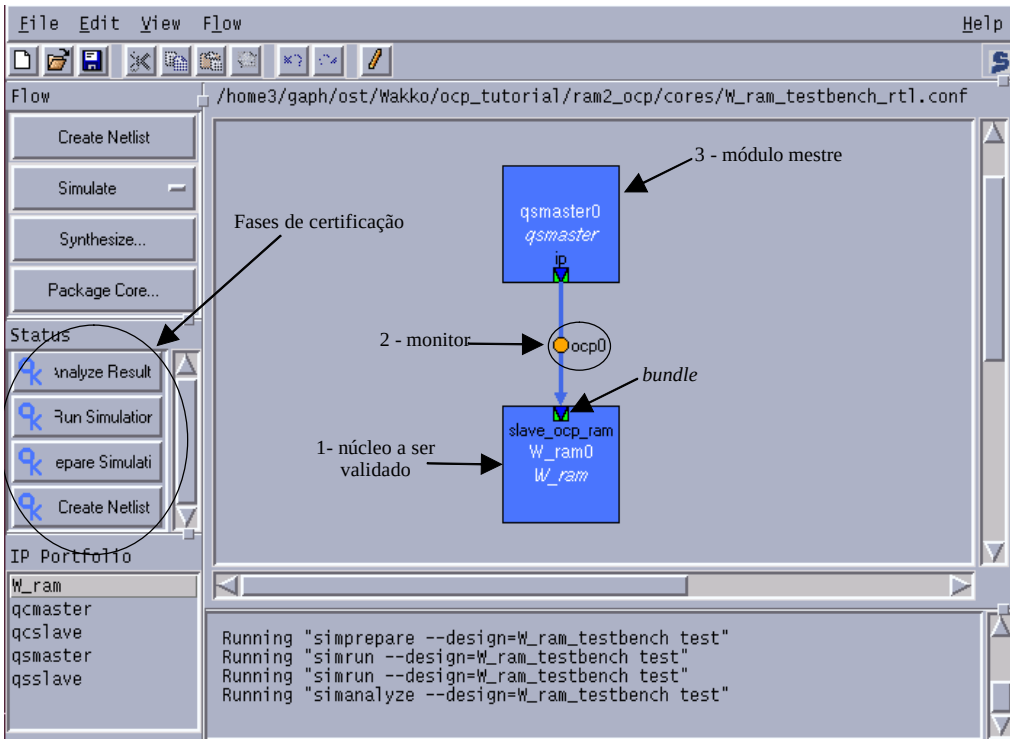


Figura 23 - Interface da ferramenta *CoreCreator* com módulos conectados entre si.

A fase denominada *criação do netlist*, é responsável por gerar a estrutura de validação do núcleo. Esta estrutura é um arquivo em formato Verilog correspondente a um *netlist* que integra todos os componentes do sistema. Este arquivo Verilog contém todo o projeto e é denominado no contexto da ferramenta *CoreCreator* de *testbench.* Antes de partir para próxima fase, o usuário deve gerar um arquivo *top* que integre o *testbench* gerado na etapa anterior com geração de relógio e *reset*. Este arquivo deve ser descrito em VHDL (nomeado de *stim.vhd*) ou Verilog (nomeado de *stim.v*).

A fase final do fluxo de certificação, apresentada neste documento, é a simulação. As etapas de *empacotamento* e *síntese* não são apresentadas [OCP02], pois não fazem parte do escopo deste trabalho. A etapa de simulação compreende três processos:

(i) *preparação da simulação* – onde são compilados arquivos descritos pelo usuário;

(ii) *simulação* – nesta etapa, utiliza-se uma ferramenta comercial (*e.g. ModelSim*) para simular os arquivos Verilog gerados pela ferramenta *CoreCreator*, assim como todos os arquivos descritos pelo usuário. Durante o processo de simulação, o simulador gera um arquivo a partir do monitor OCP (*trace file*). Este arquivo é uma tabela de valores hexadecimais capturados pelo monitor dos sinais envolvidos na comunicação entre dois lados OCP (no caso da Figura 23: *qsmaster* e *W_ram*).

(iii) *análise de resultados* – nesta etapa, a ferramenta *CoreCreator* executa três módulos: (i) *ocpdis*; (*ii*) *ocpperf*; (*iii*) *ocpcheck*. Estes módulos processam o arquivo gerado no processo de simulação e geram relatórios de saída. Dentre os três módulos destaca-se o terceiro, que verifica se o núcleo com interface OCP não violou a especificação do protocolo. Os critérios adotados pelo *ocpcheck* para verificar se uma interface OCP atende à especificação do protocolo são descritos na Seção seguinte.

3.2.5 Exemplo de criação e validação de um núcleo com interface OCP

Objetivando dominar o desenvolvimento de invólucros para núcleos não OCP realizou-se o desenvolvimento de estudos de caso de alguns núcleos, como memórias, contadores, processadores, e chaves de rede intra-chip. É importante ressaltar que os *invólucros* OCP desenvolvidos utilizam o conjunto de sinais básicos do protocolo. Dentro desse contexto, segue a descrição de uma memória com interface do tipo escravo-OCP.

A Figura 24 ilustra uma memória com invólucro OCP. O invólucro *Wrapper_OCP_Ram* empacota dois módulos: (*i*) memória RAM; (*ii*) lógica de interface, denominada *Slave_OCP_Ram*. A memória atua como escravo do sistema, ou seja, atende requisições de escrita ou de leitura de um núcleo que atue como mestre do sistema. Os sinais que compõem a interface externa da RAM são: (i) *address*; – endereço de memória; (*ii*) *data* – dado de escrita/leitura; (*iii*) *ce_n* - quando ativo habilita o acesso à memória; (*iv*) *we_n* – quando ativo habilita escrita na memória; (v) *oe_n* – quando ativo habilita a leitura da memória. O módulo *Slave_OCP_Ram* apresenta em sua interface externa todos os sinais para comunicação com a memória e os sinais utilizados para comunicação com o invólucro OCP (conjunto de sinais básicos mais o sinal *Reset_i*).

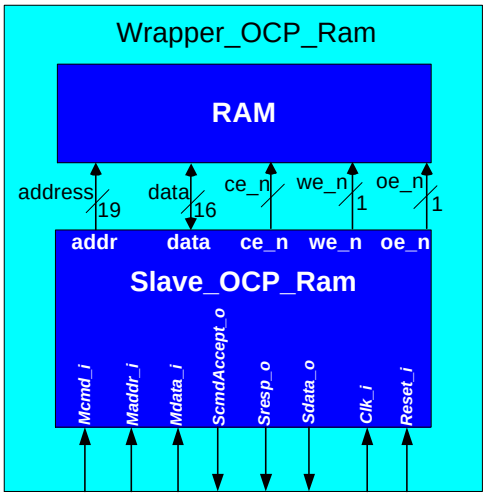


Figura 24 - *Slave_OCP_Ram* e a Ram envolvidos pelo *Wrapper_OCP_Ram*.

O comportamento responsável pelo controle do protocolo OCP, implementada no módulo *Slave_OCP_Ram*, é descrito na Figura 25.

A máquina de estados, responsável pela interface entre a RAM e protocolo OCP, apresenta os seguintes estados:

IDLE: neste estado, o módulo *Slave_OCP_Ram* fica aguardando até receber uma requisição de escrita ou leitura do módulo que atua como mestre do sistema. Ao receber uma requisição

válida (WR/RD) passa-se para o estado correspondente à requisição.

WRITE: os sinais *ce_n* e *we_n* são ativados. No mesmo estado, o sinal *address* recebe *Maddr_i*, ou seja, o endereço da memória no qual o dado deve ser escrito e o *data* recebe *Mdata_i*.

READ: estado de leitura, ativam-se os sinais *ce_n* e *oe_n* e transfere-se para o sinal *address* o conteúdo do *Maddr_i*, o qual corresponde ao endereço de memória para leitura. Neste mesmo estado, é informado ao mestre que o dado está disponível (*Sresp_o* <= DVA) e o dado lido é transferido para o sinal *Sdata_o*.

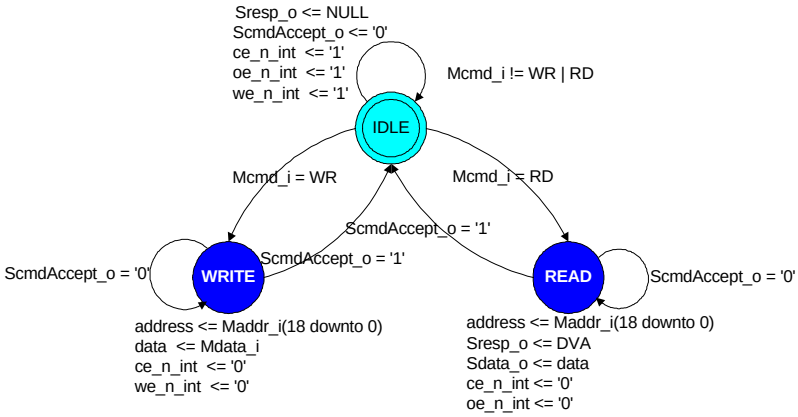


Figura 25 - Máquina de estados do módulo *Slave_OCP_Ram*.

Após o desenvolvimento do invólucro OCP torna-se necessário validar o mesmo. A estrutura de validação adotada é a mesma ilustrada na Figura 23 (página 35) . A Figura 26 ilustra os resultados da simulação para requisições de escrita e leitura geradas pelo módulo *QS-Master* para memória com interface OCP. A seqüência de eventos destacada na Figura 26 é descrita abaixo:

1. A fase de requisição ocorre com a transição de IDLE para WR no sinal *Mcmd_i* = ‘1’. No mesmo instante os sinais *Maddr_i* e *Mdata_i* recebem do módulo mestre (*QS-Master*) o endereço de memória e o dado a ser escrito, respectivamente.

2. Um ciclo de relógio depois da requisição de escrita, o escravo (RAM) ativa os sinais *ce_n* e *we_n*. Neste momento, o sinal *ScmdAccept_o* é ativado, indicando ao mestre que a escrita do dado (CDEF) no endereço de memória informado no sinal *address* foi feita. A latência gerada na fase de escrita é igual a 1 ciclo de relógio.

3. Após ficar um ciclo de relógio em IDLE o módulo escravo recebe uma requisição de leitura (*Mcmd_i* = ‘2’). No mesmo instante o campo *Maddr_i* recebe o endereço que será utilizado para leitura.

4. Um ciclo de relógio depois da requisição de leitura, os sinais *ce_n* e *oe_n* são ativados. No mesmo ciclo, *ScmdAccept_o* é ativo e o endereço de memória é recebido no sinal *address*.

5. Após um ciclo de relógio, é apresentada uma resposta válida no sinal *Sresp_o*. No mesmo instante, o dado é colocado no sinal *Sdata_o*. A latência gerada por essa fase é igual a 2 ciclos de relógio.

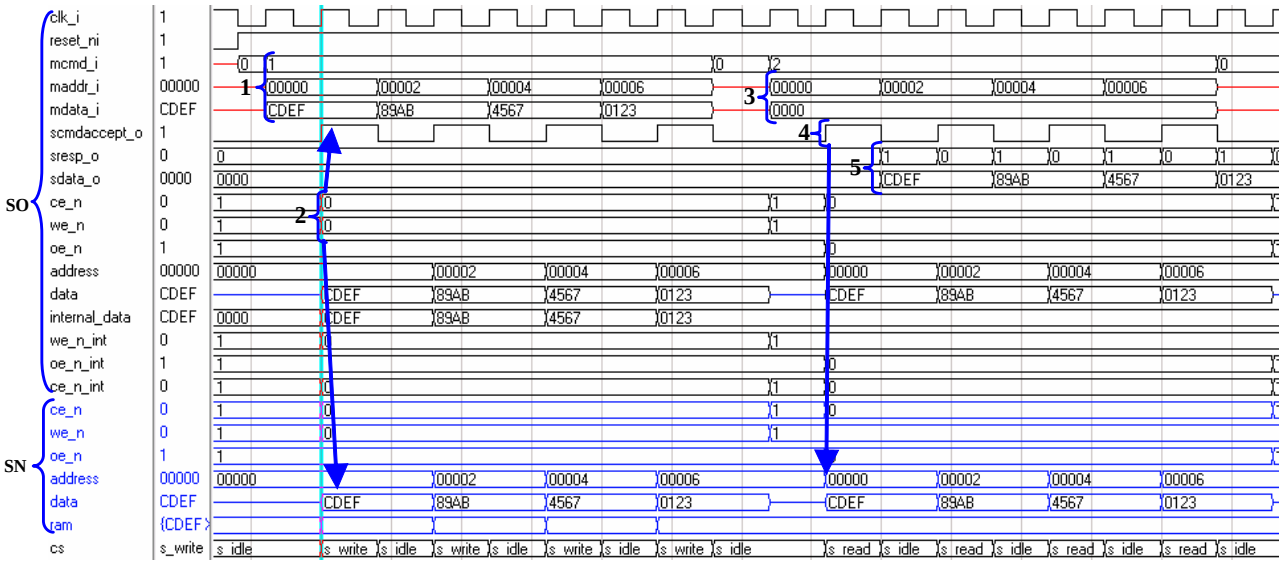


Figura 26 - Simulação para requisição de escrita e leitura. SO indica os sinais do módulo *Slave_OCP_RAM* e DN os sinais do módulo RAM.

A execução desta simulação, quando lançada pela ferramenta *CoreCreator*, fornece relatórios referentes a transações e tráfego OCP, e se estes estão conforme o protocolo em questão. Exemplos de relatórios gerados para o núcleo validado acima são ilustrados nas Figura 27 e Figura 28. Estando de acordo com o protocolo, diz-se que o núcleo é compatível com OCP.

```
  Check: ocp0_ocpmon                      Status
---------------------------------------- ----------
Address Alignment                        PASS (1)
Complete Transfers                       PASS (2)
Control/Status Handshaking               PASS (3)
Group Signal Integrity                   PASS (4)
Protocol                                 PASS (5)
Reset                                    PASS (6)
Signal Integrity                         PASS (7)
```

Figura 27 - Exemplo de relatório quanto à compatibilidade com OCP, gerado pelo módulo *ocpcheck* da ferramenta *CoreCreator*.

Para determinar se uma interface atende as normas OCP, o *ocpcheck* verifica uma série critérios para atribuir *PASS* aos itens ilustrados na Figura 27. A descrição desses critérios segue:

(1) *Address Aligment*: verifica se os endereços estão alinhados corretamente durante as transações.

(2) *Complete Transfers*: os dados que são escritos em um conjunto de endereços são comparados com os dados lidos do mesmo conjunto de endereços. Se estes não forem idênticos, considera-se que este núcleo não atende as normas OCP neste quesito.

(3) *Control/StatusHandshaking*: verifica-se se os sinais de *handshake* da interface mudam de acordo as especificações do protocolo.

(4) *Group Signal Integrity*: existem sinais que devem ser ativados no mesmo ciclo de relógio, como, *Mcmd* e *Mdata* devem receber valores no mesmo instante.

(5) *Protocol*: se todas as fases de uma transação OCP (requisição, resposta e *handshaking* de

dados se configurado) ocorreram de acordo com o que rege o protocolo atribui-se *PASS* para este item.

(6) *Reset*: atribui-se *PASS* a este item se o sinal *reset* foi ativado depois de 16 ciclos de relógio e se os dados recebem valores de acordo com o estado desse sinal;

(7) S*inal integrity*: se nenhum sinal OCP exibe valores como 'X' ou 'Z' durante a simulação, considera-se que estes sinais seguem o que rege o protocolo.

```
SimTime   Cycle  CAD Cmd       Addr  Data  Resp RVD
--------- ------ --- --------- ----- ----- ---- ---
     5300     26   1     WRITE 00000  cdef        .
     5700     28   1     WRITE 00002  89ab        .
     6100     30   1     WRITE 00004  4567        .
     6500     32   1     WRITE 00006  0123        .
     7100     35   1      READ 00000  cdef DVA    2
     7500     37   1      READ 00002  89ab DVA    2
     7900     39   1      READ 00004  4567 DVA    2
     8300     41   1      READ 00006  0123 DVA    2
```

Figura 28 - Exemplo de relatório quanto ao ciclos OCP gerados a partir do tráfego especificado em STL, gerado pelo *ocpdis* da ferramenta *CoreCreator*.

O relatório da Figura 28 prove de maneira sucinta as transações ocorridas, assim como os sinais e os valores envolvidos durante as transações. Por exemplo, no tempo de simulação 5700 ns, ocorre a 28ª transação OCP, correspondendo a uma operação de escrita no endereço 00002 com o valor 89AB. Quando ocorrem transações de leitura, o sinal de resposta DVA é recebido.

3.3 Considerações finais do Capítulo

Após a breve descrição dos protocolos VCI e OCP, torna-se necessário salientar alguns pontos em comum entre eles. Ambos os protocolos baseiam-se na abordagem de comunicação ponto-a-ponto. Tanto VCI como OCP possuem capacidades similares que permitem interconectar núcleos num mesmo CI com menor esforço. Porém, existem diferenças entre estas abordagens. O conjunto de sinais de teste que compõe OCP pode ser considerado um diferencial, já que VCI não suporta algo semelhante. Além do conjunto de sinais de teste, o OCP possui um ferramental que permite a validação e certificação de núcleos OCP que venham a ser desenvolvidos pelo usuário.

Além disso, foi anunciado recentemente que a OCP-IP e a VSIA pretendem unificar o protocolo para interconexão de núcleos [WIL03].

O conjunto de fatores que levaram à escolha de protocolo OCP pode ser resumido em: (*i*) ser um protocolo padrão aceito pela indústria; (*ii*) ter ferramental para certificação de núcleos; (*iii*) possível padrão *de fato* para interconexão de núcleos com a unificação OCP-VSIA; (*iv*) uso do protocolo OCP no projeto de pesquisa Brazil-IP[1], projeto este onde a presente dissertação está inserida.

[1] http://www.brazilip.org/fenix/index.asp

4 REDE HERMES-OCP

Este Capítulo descreve a rede intra-chip, denominada HERMES, desenvolvida no grupo de atuação de pesquisa do autor dessa dissertação. A HERMES foi utilizada como base para o desenvolvimento das atividades que compõe essa dissertação. Após uma descrição sucinta da HERMES apresenta-se a primeira contribuição deste trabalho, Seção 4.2, a qual corresponde ao desenvolvimento de interfaces de rede OCP para a rede intra-chip em questão.

4.1 Características da Rede HERMES

A rede HERMES é uma infra-estrutura para implementação de redes intra-chip. Diz-se infra-estrutura porque não existe uma única rede intra-chip. Existe um conjunto de módulos que podem ser utilizados para geração de redes intra-chip. Exemplos desses módulos são: árbitro, filas e portas de entrada/saída. A descrição detalhada de cada módulo encontra-se disponível em [MEL03]. Todos os módulos da HERMES são parametrizáveis, através de uma ferramenta que permite a geração de redes intra-chip em função das restrições de projeto como: (*i*) largura de palavra; (*ii*) profundidade das filas; (*iii*) topologia da rede, entre outras. No Capítulo 5, esta ferramenta é apresentada detalhadamente.

Utilizando os módulos citados acima é possível gerar redes intra-chip diretas, com topologias como malha e torus. Existem protótipos funcionais para a topologia malha implementados no grupo de atuação [MOR03].

A HERMES implementa os três níveis hierárquicos inferiores do modelo de referência OSI: (*i*) físico; (*ii*) enlace e (*iii*) rede. O nível físico da rede HERMES implementa a interface de comunicação entre as chaves. O nível de enlace adota o protocolo de *handshake* para o envio e recebimento de dados de forma confiável. Para compor o nível de rede é adotado o modo de chaveamento de pacotes *wormhole*. O pacote na rede HERMES é formado por 1 *flit* com endereço destino, 1 *flit* com o tamanho do *payload* e de 0 a 2^{n-1} *flits* de *payload*, onde *n* corresponde ao tamanho do *flit*. A rede HERMES utiliza roteamento distribuído, determinístico e com caminho mínimo entre os nodos de origem e destino.

4.1.1 Chave HERMES

A chave HERMES possui uma lógica de controle de chaveamento centralizada e cinco portas bidirecionais parametrizáveis: (*i*) norte; (*ii*) sul; (*iii*) leste; (*iv*) oste; e (*v*) local. A estrutura básica de uma chave HERMES é apresentada na Figura 29. Cada porta (ou canal) é constituído por um conjunto sinais que implementam o protocolo de *handshake*. A funcionalidade destes sinais é a seguinte:

- *data_out*: dado de saída do canal, com largura configurada de acordo com a aplicação (8, 16, 32 bits);
- *tx*: indica a presença de um dado válido na porta de saída do canal, enviado pelo transmissor do dado;

- *ack_tx*: reconhecimento de recepção do dado válido, enviado pelo receptor do dado;
- *data_in*: dado de entrada da porta, com largura configurável;
- *rx*: indica a presença de um dado válido na porta de entrada enviado pelo transmissor do dado;
- *ack_rx*: reconhecimento de recepção do dado válido, enviado pelo receptor do dado.

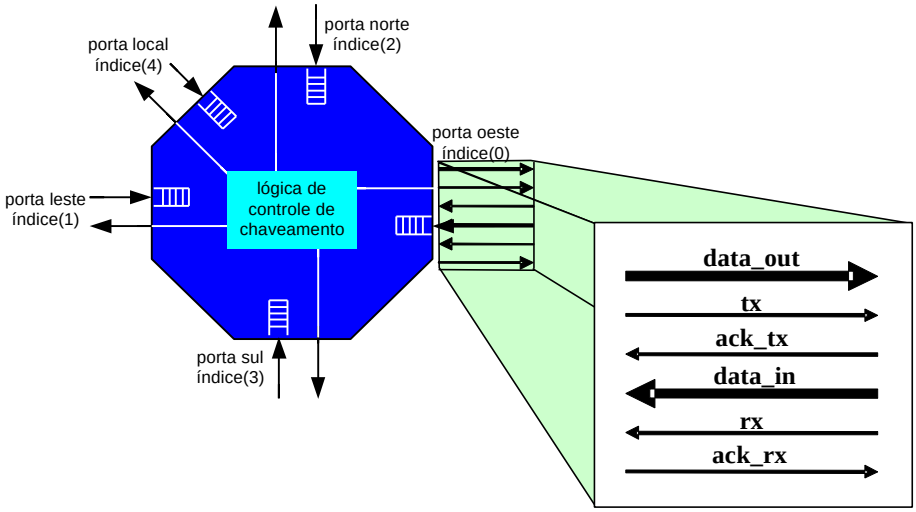


Figura 29 - Arquitetura da chave HERMES.

Para topologias malha e torus, no máximo cinco portas de acesso são necessárias, sendo uma porta local para conexão ao núcleo e quatro portas para conexão a outras chaves da rede. No caso de topologias torus todas as chaves possuem 5 portas. Na topologia malha, o número de portas depende da posição da chave relativa aos limites da malha. Neste caso, uma mesma topologia permite a parametrização da chave quanto ao número de portas e filas em função da posição da chave. A ferramenta desenvolvida para geração automática da rede realiza esta otimização. Na Seção 5.1.3, página 70, apresenta-se um exemplo da redução de área obtida com esta otimização.

Outra particularidade inerente à chave da HERMES foi a de limitar o esquema de armazenamento de *flits*, empregando apenas filas de entrada (*input buffering*), descartando as possibilidades de estocagem na saída da chave (filas de saída) e/ou esquemas mistos de armazenamento. Isto foi feito porque o armazenamento na entrada da chave é aquele que causa a mínima sobrecarga de área, e este parâmetro, como atestado na bibliografia disponível, determina a área ocupada pela chave. O tamanho da área de estocagem é também parametrizável. Resultados de análises iniciais visando determinar o tamanho ótimo dos *buffers* de entrada na procura do máximo desempenho para a chave HERMES com a mínima área, encontram-se disponíveis em [MOR03a].

4.1.1.1 Validação Funcional da chave HERMES

A chave da HERMES é descrita em VHDL e validada por simulação funcional. Alguns dos blocos internos da chave e os sinais de duas portas (local e leste) são ilustrados na Figura 30. A Figura 31 apresenta a simulação funcional dos sinais mais importantes da chave HERMES.

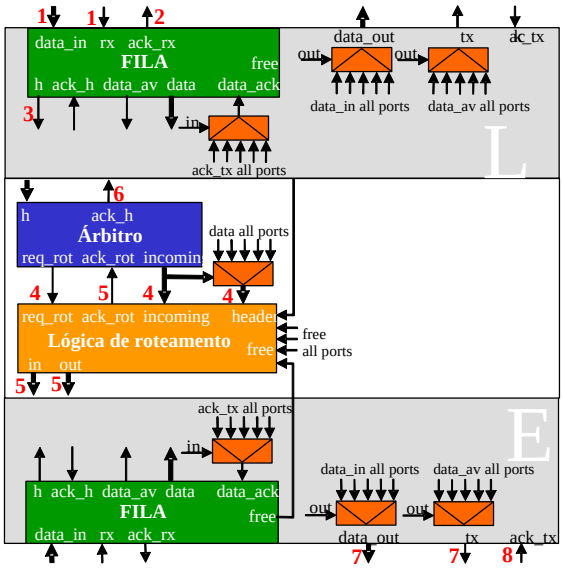


Figura 30 - Diagrama de blocos parcial da chave, mostrando duas de cinco portas. Os números na figura correspondem à seqüência de eventos apresentados na Figura 31.

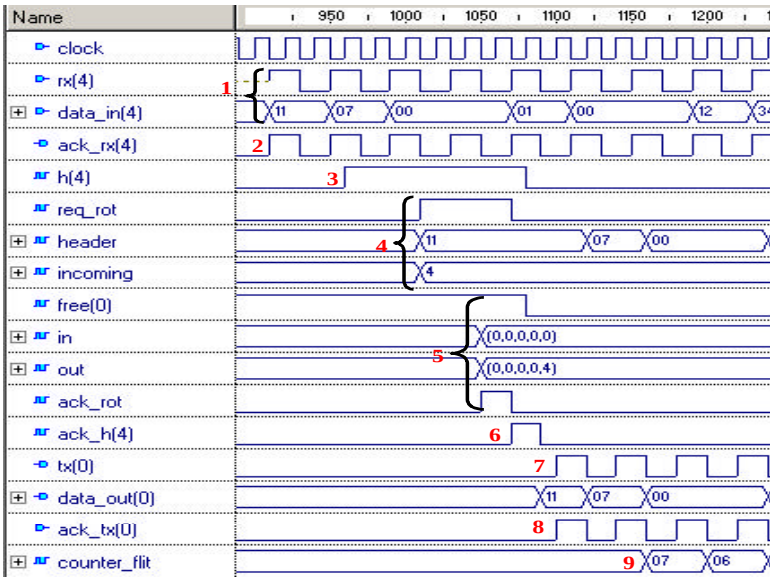


Figura 31 - Simulação de uma conexão entre a porta local e a porta leste.

Os passos da simulação são descritos a seguir:

1. A chave recebe um *flit* da porta local (índice 1 na Figura), o sinal *rx* é ativo e o sinal *data_in* recebe o conteúdo do *flit*.

2. O *flit* é armazenado no buffer local e o sinal *ack_rx* é ativo indicando que o *flit* foi recebido (índice 2 na Figura).

3. A porta local solicita roteamento para ao árbitro (lógica de arbitragem) ativando o sinal *h* (índice 3 na Figura).

4. O árbitro pode receber múltiplas solicitações de roteamento (*h*) simultâneas. Uma vez determinada a porta de entrada que será atendida, o árbitro solicita à lógica de roteamento conexão a uma dada saída. Isto é feito ativando-se o sinal *req_rot* (índice 4 na Figura). O

sinal *header* contém o destino (valor 11), e o sinal *incoming* a origem do pacote (4, que indica a porta local).

5. O algoritmo de roteamento é executado, a tabela de roteamento é escrita e o sinal *ack_rot* é ativado (índice 5 na Figura), indicando o estabelecimento da conexão, quando possível. Caso não seja possível rotear a solicitação, o árbitro seleciona a próxima porta com pacotes a transmitir.

6. A lógica de arbitragem informa o buffer que a conexão foi estabelecida e que o *flit* pode ser transmitido (índice 6 na Figura).

7. A chave ativa o sinal *tx* da porta de saída selecionada e coloca o conteúdo do *flit* no sinal *data_out* da mesma porta (índice 7 na Figura).

8. Uma vez que o sinal *ack_tx* está ativo o *flit* é removido do buffer e o próximo *flit* armazenado é enviado.

9. O segundo *flit* inicializa o contador, indicando quantos *flits* devem ser enviados para que a conexão ser finalizada (no presente caso, 7 *flits*).

A chave, como apresentado na Figura 31, apresenta uma latência de 8 ciclos de relógio para rotear o cabeçalho (primeiro *flit*), se não houverem colisões ou portas de saídas ocupadas. Após essa latência, os *flits* subseqüentes são roteados a cada dois ciclos de relógio, devido ao protocolo *handshake*.

Considerando que uma chave pode ter até cinco conexões simultâneas, e sendo a freqüência de relógio 25 MHz (estabelecida por propósitos de roteamento), e a largura do *flit* igual a 8 bits, a chave pode suportar um pico de taxa de transmissão igual a 500 Mbps (5 * 25 * 8 / 2). Divide-se o produto por dois, devido ao fato do protocolo *handshake* necessitar 2 ciclos de relógio para transmitir um *flit*.

4.1.2 Interface de Rede da HERMES

A interface de rede original da Hermes é denominada Send/Receive [MEL03]. Esta IR é um módulo quem contém a lógica de envio e recebimento de pacotes, assim como a montagem, o envio e o recebimento de confirmações. Esta IR foi implementada em hardware, do lado do núcleo. A Figura 32 ilustra este módulo e seus respectivos sinais são detalhados a seguir.

- *avSend*: sinaliza que o núcleo tem um pacote a ser enviado através do módulo Send.
- *nWordSend*: número de palavras do pacote a ser enviado.
- *dataSend*: vetor correspondente ao pacote a ser enviado. Este vetor pode conter até 8 palavras de 16 bits.
- *busySend*: sinaliza que o módulo Send está transmitindo pacote.
- *avReceive*: indica que o módulo Receive recebeu pacote destinado ao núcleo.

- *dataReceive*: vetor correspondente ao pacote recebido.
- *busyCore*: indica que o núcleo não pode receber pacotes no momento.
- *addressCore*: endereço do núcleo ao qual o módulo Send/Receive está vinculado.

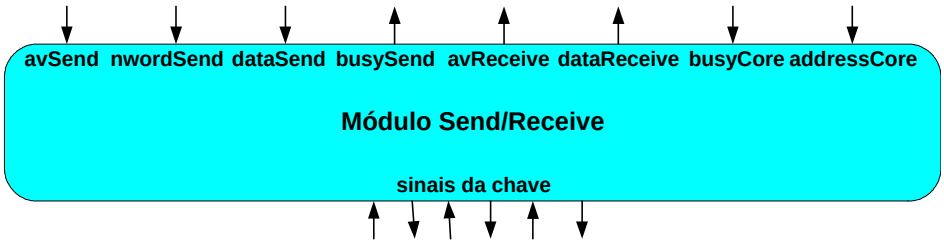


Figura 32 - Interface do módulo Send/Receive.

O módulo Send/Receive possui duas máquinas de estados: (*i*) máquina de estados *Send*; e (*ii*) máquina de estados *Receive*.

A máquina de estados *Send* fragmenta os pacotes em *flits* para que os mesmos possam ser enviados pela HERMES. Por sua vez, a máquina de estados *Receive* é responsável pela recepção dos *flits* destinados ao núcleo vinculado ao módulo Send/Receive.

Esta interface de rede, apesar de funcional, não atende ao requisito de reusabilidade, pois é implementada no lado do núcleo, sendo também proprietária. Além disto, o consumo de área é elevado devido aos vetores utilizados para recepção/envio de dados.

4.1.3 Validação da Rede HERMES

A rede HERMES foi validada com a topologia malha. A Figura 33 ilustra a transmissão de um pacote da chave 00 para chave 11 numa topologia malha. Deve-se ressaltar, que apenas os comportamentos de entradas e saídas da chave 10 são ilustrados na simulação.

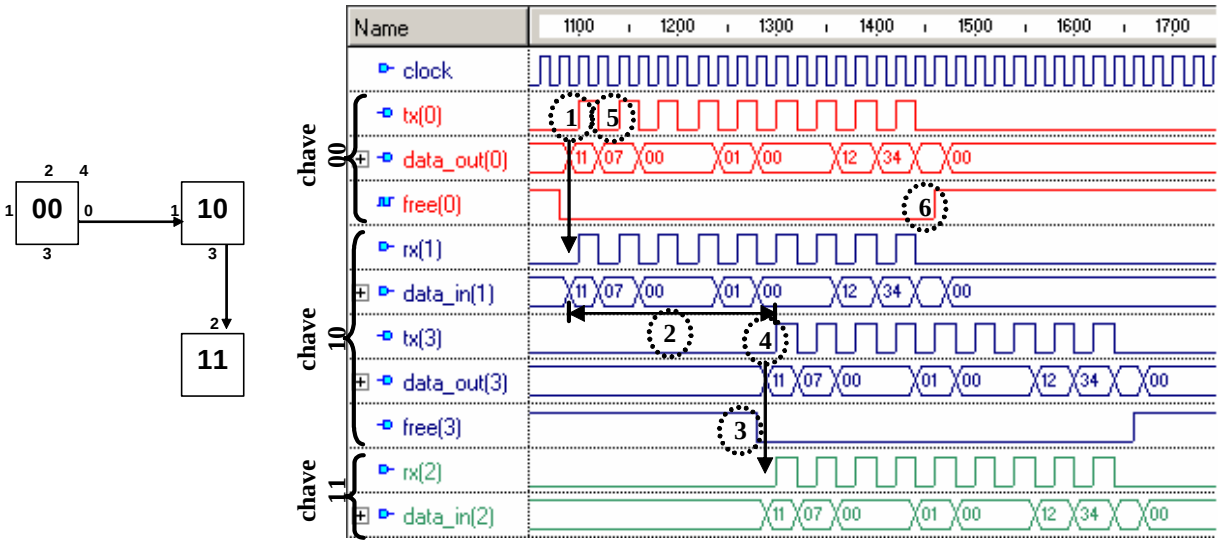


Figura 33 - Simulação da transmissão de um pacote da chave 00 para chave 11 em uma topologia malha de dimensão 2x2.

Os passos da simulação são descritos a seguir:

1. A chave 00 envia o primeiro *flit* do pacote (endereço de destino) para o sinal *data_out* da porta leste (índice 0) e ativa o sinal *tx* da mesma porta.

2. A chave 10 detecta que o sinal *rx* está ativo na porta oeste (índice 1) e captura o *flit* do sinal *data_in* São necessários 10 ciclos de relógio para rotear esse *flit*. Os próximos *flits* são roteados com dois ciclos de relógio de latência.

3. A porta de saída sul (índice 3) da chave 10 está livre, *free*(3).=‘1’. Este sinal passa para ‘0’, indicando ocupação da porta.

4. A chave 10 coloca o *flit* no sinal *data_out* e ativa o sinal *tx* da porta sul. Depois, a chave 11 detecta o sinal *rx* da porta norte ativo. O *flit* é capturado no sinal *data_in* e a conexão entre o fonte e o destino é estabelecida.

5. O segundo *flit* do pacote contém o número de *flits* que compõem o *payload.*

6. Depois que todos os *flits* são enviado, a conexão é finalizada e todas a entradas do vetor *free* de cada chave envolvida na conexão retornam aos seus estados de *free* (igual a ‘1’).

A latência mínima (ciclos de relógio) para transferir um pacote do fonte ao seu destino é dado por:

$$latência = \left(\sum_{i=1}^{n} R_i\right) + P \times 2$$

onde n é o número de chaves envolvidas na comunicação (inclusive a fonte e o destino), R_i é o tempo necessário pelo algoritmo de roteamento em cada chave visitada (10 ciclos de relógio), e P é o tamanho do pacote. Esse número é multiplicado por 2 porque cada *flit* necessita de 2 ciclos de relógio para ser enviado.

4.2 Interfaces de Rede OCP e protocolo de comunicação para HERMES

Como descrito anteriormente, uma estrutura de interconexão deve atender ao requisito de reusabilidade de projeto. Visando aumentar a reusabilidade da HERMES em projetos distintos, padronizou-se as interfaces de rede com o protocolo OCP. Denomina-se HERMES-OCP a rede intra-chip HERMES com interface OCP. *Objetiva-se* com essa proposta simplificar ao máximo as transações (núcleo-chave), tornando a HERMES o mais transparente possível para os núcleos que a empregam como meio de comunicação.

Dependendo do núcleo, a porta local pode possuir uma interface de rede OCP do tipo mestre, escravo ou mestre-escravo. Núcleos que desejem comunicar-se via a HERMES-OCP devem atender ao protocolo OCP *adotado* pela rede. Não basta que os núcleos possuam interfaces OCP compatíveis. Deve-se definir sobre o protocolo OCP o conjunto de sinais utilizados, e qual o protocolo de comunicação adotado pela rede. Em outras palavras, o nível físico da HERMES adota OCP, o nível de enlace define um formato de pacote, composto por uma seqüência de transações OCP e o nível de rede é uma comunicação ponto-a-ponto, que pode ser diferente para cada par de

núcleos. Neste contexto foram desenvolvidos para a rede HERMES três tipos de IRs OCP: (*i*) IR mestre (IR-M); (*ii*) IR escravo (IR-E); e (*iii*) IR mestre-escravo (IR-ME).

Até o presente momento, as interfaces de rede propostas suportam apenas transações de escrita e leitura. Núcleos que queiram escrever (enviar pacotes) ou ler dados (receber pacotes) para/de outros núcleos através da HERMES-OCP devem atender ao protocolo de comunicação da rede, como proposto abaixo:

- não utilizar o sinal *Maddr* para transmitir dados através da HERMES-OCP, pois na rede não há transmissão separada de dados e endereços. Assim, endereços devem ser encapsulados da mesma forma que informação útil (dados). Por este motivo, a largura desse sinal é reduzida a um bit (largura mínima na interface OCP da rede HERMES). Utiliza-se esse sinal para cumprir a especificação mínima de sinais básicos OCP, conforme descrito em [OCP02].

- independente da operação desejada (leitura ou escrita) as duas primeiras transações devem transmitir, respectivamente, o endereço do núcleo destino e o número de palavras de *N* bits (onde *N* corresponde a 8, 16, 32 e 64) que compõe o restante do *payload*.

- não é possível misturar comandos de escrita e leitura em um pacote enviado pela rede HERMES-OCP.

- o tamanho máximo do *payload* é de 2^N-1 palavras de *N* bits, onde *N* corresponde ao tamanho do *flit* em bits. Caso um determinado núcleo necessite um pacote maior que este, é responsabilidade da IR do mesmo realizar os processos de segmentação e remontagem. Note-se que isto adiciona uma camada ao protocolo de comunicação da HERMES-OCP, sob responsabilidade do usuário.

Tanto para transações de envio como para de recepção, um conjunto de transações básicas OCP torna-se necessário. Por exemplo, para o núcleo mestre escrever dados em um núcleo escravo, as transações descritas na Tabela 5 devem ser executadas.

Tabela 5 - Protocolo para transmissão de dados (escrita) na rede HERMES.

Sinal *MData*	**Destino**	**Tamanho**	**d_0**	**d_1**	**...**	**d_{n-1}**
Sinal *MCmd*	**WR**	**WR**	**WR**	**WR**	**WR**	**WR**

Para um núcleo mestre ler dados de outro núcleo, este deve gerar as transações descritas na Tabela 6:

Tabela 6 - Protocolo para transmissão de leituras na rede HERMES.

Sinal *MData*	**Destino**	**Tamanho**	**d_0**	**d_1**	**...**	**d_{n-1}**
Sinal *MCmd*	**WR**	**WR**	**RD**	**RD**	**RD**	**RD**

Para um núcleo escravo responder às solicitações de leitura providas por um núcleo mestre, este deve gerar as transações descritas na Tabela 7.

Tabela 7 - Protocolo para transmissão de respostas de dados lidos na rede HERMES.

Sinal *SData*	**d_0**	**d_1**	**...**	**d_{n-1}**
Sinal *SResp*	**DVA**	**DVA**	**DVA**	**DVA**

Cabe ressaltar, que este protocolo de comunicação proposto adota um modelo de comunicação do tipo NORMA (*No-Remote Memory Access*). Neste modelo não há um mapa global de endereços, sendo a comunicação realizada pela troca de mensagens entre os núcleos. Por exemplo, caso um determinado núcleo deseje escrever um conjunto de dados na memória, a mensagem que este enviará para a memória será o endereço da mesma na rede, o endereço inicial de escrita e logo após os dados.

Uma alternativa a este modelo, adotada no projeto Brazil IP, é o modelo NUMA (do inglês, *Non-Uniform Memory Access*). Neste modelo há um espaço único de endereçamento, sendo muito similar às estruturas tradicionais de barramento. Este modelo pode parecer atrativo em um primeiro momento, pois simplifica a comunicação, pois basta enviar o endereço de destino e o dado. Entretanto, este modelo apresenta desvantagens que são descritas abaixo:

- mensagens são tipicamente curtas, compostas normalmente por um endereço e um dado. Redes intra-chip terão seu desempenho maximizado no momento que se transfere dados em rajada.

- reuso dos núcleos é comprometido, pois cabe ao projetista definir o mapa de memória e evitar conflitos entre os núcleos. Em um sistema NORMA cada núcleo é independente, bastando haver a informação dos "serviços" que este pode oferecer aos demais núcleos.

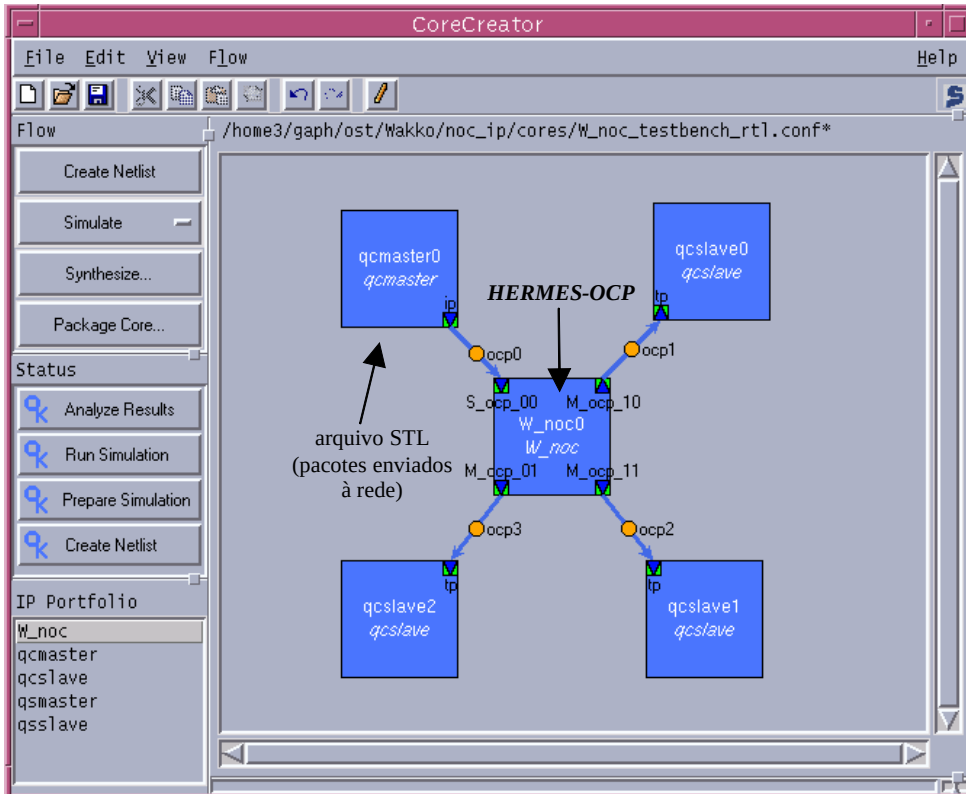
- o modelo NUMA implica no envio do par endereço/dado. Freqüentemente, o endereço é dispensável, pois transferências de dados vão ocorrer em blocos (multimídia e comunicação externa), bastando a informação do endereço inicial da transferência.

As desvantagens citadas acima guiaram a escolha do projeto das IRs OCP e dos sistemas desenvolvidos no contexto dessa dissertação para o modelo NORMA. Objetiva-se na Seção seguinte apresentar o desenvolvimento e a validação das IRs OCP citadas anteriormente. Essas IRs foram descritas em VHDL, validadas por simulação funcional e prototipadas em FPGA (este processo é discutido no Capítulo 6 deste trabalho).

4.2.1 Estrutura de Validação para IRs via *CoreCreator*

Foi desenvolvida uma estrutura de validação sobre uma rede HERMES com topologia malha de dimensão 2 x 2, para a certificação, via *CoreCreator*, da HERMES-OCP. Conectou-se a três portas locais dessa rede IRs-M (chaves de endereços: 10, 11 e 01) e na porta restante uma IR-E (chave de endereço 00). Por conseqüente, foram conectados módulos *QC-Slave* às IRs-M e um módulo *QC-Master* à IR-E. A estrutura desenvolvida para validar as IRs OCP da HERMES via

CoreCreator é ilustrada Figura 34.



A figura acima ilustra a estrutura de validação utilizada para validar as IRs da HERMES-OCP de dimensão 2x2 (representado na figura como W_noc). As três IRs-M são representadas pelos bundles: M_ocp_10 (porta local da chave 10); M_ocp_01 (porta local da chave 01); M_ocp_11 (porta local da chave 11). Já a IR-E é representada pelo bundle (S_ocp_00). O módulo QC-Master é responsável por gerar os estímulos OCP (pacotes) que devem exercitar as interfaces de rede OCP. Estes estímulos foram descritos em STL conforme ilustram as Figura 35 e Figura 36. Além desses módulos compõem a estrutura 4 monitores OCP que devem verificar as transações entre os módulos QC-Master/Slave e as IRs do lado da rede (W_noc) .

Figura 34 - Estrutura de validação para o estudo de caso via *CoreCreator*.

O módulo *QC-Master* é responsável pelo envio de pacotes à rede. Para isso, foi descrito um arquivo em STL que contém 2 pacotes de 128 bits destinados à mesma chave (chave de endereço 11). Para que um pacote possa ser transferido pela HERMES-OCP deve-se fragmentar o mesmo em *flits*. Um dos pacotes contém *flits* que devem ser enviados a rede (transações de escrita OCP). A descrição STL e a fragmentação deste pacote de 128 bits é ilustrada na Figura 35.

Pacote de 128 bits descrito em STL

Write128 0x0 0x0005000400030002000100000060011

Fragmentação do pacote de 128 bits em 8 *flits* (escrita)

flit 1	*flit* 2	*flit* 3	*flit* 4	*flit* 5	*flit* 6	*flit* 7	*flit* 8
0011	0006	0000	0001	0002	0003	0004	0005
WR	WR	WR	WR	WR	WR	WR	WR

Figura 35 - Fragmentação de um pacote de 128 bits em *flits*.

O segundo pacote, descrito em STL, utilizado para transmitir leituras pela HERMES-OCP é ilustrado na Figura 36.

Pacote de 128 bits descrito em STL para transmitir leituras

```
write32 0x0 0x00060011
read16 0x0 0x0000
read16 0x0 0x0001
read16 0x0 0x0002
read16 0x0 0x0003
read16 0x0 0x0004
read16 0x0 0x0005
```

Fragmentação do pacote de 128 bits em 8 *flits* (leitura)

flit 1	*flit* 2	*flit* 3	*flit* 4	*flit* 5	*flit* 6	*flit* 7	*flit* 8
0011	0006	0000	0001	0002	0003	0004	0005
WR	WR	RD	RD	RD	RD	RD	RD

Figura 36 - Descrição STL de um pacote de 128 bits contendo requisições de leitura.

As Seções subseqüentes apresentam o projeto das IRs e a validação das mesmas (4.2.2, 4.2.3, 4.2.4). O resultado da certificação das IRs frente ao protocolo OCP é apresentado na 4.3.

4.2.2 Interface de rede escravo OCP

As interfaces de rede na NoC do tipo escravo permitem a conexão de núcleos com interface de rede do tipo mestre (ver na Figura 34 o módulo *qcmaster0* conectado na IR-E *S_ocp_00*). Os sinais OCP que compõem a IR-E são descritos na Tabela 8.

Tabela 8 - Sinais OCP da Interface Escravo.

Grupo	Sinal	Largura(bits)	*Driver*
Básico	SClk_i	1	Mestre/Escravo
	MAddr_i	1	Mestre
	MCmd_i	3	Mestre
	MData_i	8, 16, 32 ou 64	Mestre
	SCmdAccept_o	1	Escravo
	SData_o	8, 16, 32 ou 64	Escravo
	SResp_o	2	Escravo
Fluxo de dados	MrespAccept_i	1	Mestre
Opcionais de controle	SReset_ni	1	Mestre / Escravo

A lógica responsável pela segmentação e remontagem de pacotes foi dividida em duas máquinas de estado: (*i*) **máquina de recepção *núcleo* -** máquina responsável pela recepção de requisições OCP provindas da IR-M do núcleo e montagem do pacote para envio pela rede, conforme descrito na Figura 38; e (*ii*) **máquina de recepção *chave* -** máquina responsável pela recepção e montagem dos pacotes provindos da rede para a IR-M do núcleo, como ilustrado na Figura 39.

Na **máquina de recepção *núcleo*** observa-se que a montagem do pacote inicia quando a IR-E recebe uma requisição de escrita (conforme definido no protocolo de comunicação). Para cada *flit* recebido pela IR-E do núcleo mestre são necessários dois estados: um para capturar e informar que existe um *flit* a ser transmitido (*txL* = “1”) e outro para receber a confirmação que o *flit* foi transmitido à porta local da chave. O formato do pacote gerado por essa máquina é ilustrado na

Figura 37.

flit 1	*flit* 2	*flit* 3	*flit 4*	...	*flit* n
endereço	tamanho do pacote + 1	comando OCP	dado/ endereço	...	dado/ endereço

Figura 37 - Formato do pacote gerado pela máquina de estados de recepção, com dados oriundos do núcleo mestre.

O *flit1* recebe o endereço do destino ao qual o pacote deve ser entregue. No segundo *flit* adiciona-se 1 ao valor que corresponde ao tamanho do pacote, isto porque um comando OCP deve ser inserido para informar a transação a ser efetuada na interface OCP do núcleo destino (leitura ou escrita). Conseqüentemente, o terceiro *flit* do pacote recebe um comando OCP. Finalmente, os *flits* restantes recebem os dados a serem escritos ou os endereços de dados a serem lidos.

A descrição dos estados que compõem a *máquina de recepção núcleo* (Figura 38) é descrita abaixo:

IDLE: neste estado, o módulo IR-E OCP fica aguardando até receber uma requisição de escrita do núcleo com interface mestre-OCP. Ao receber uma requisição válida (WR/RD) e se o sinal *busyMaster* não estiver ativo passa-se para o estado SS1. O sinal *busyMaster*, se ativo, indica ao módulo IR-E OCP que a porta local da chave esta sendo utilizada pelo módulo IR-M. Também neste estado, todos os sinais de saída deste módulo são inicializados em ZERO.

SS1: no presente estado, inicializa-se a montagem do pacote que será enviado pela rede (*flit 1*). No mesmo estado, o sinal *txL* é ativo e o sinal *data_outL* recebe os endereços fonte e destino. Quando o sinal *ack_txL* é ativo passa-se para o próximo estado.

SS2: o sinal *auxScmdAccept* é ativo e o sinal *txL* é desativado. Quando o sinal *Mcmd_i* receber um comando válido e o sinal *auxScmdAccept* for desativado passa-se para o estado SS3.

SS3: neste estado, ativa-se o sinal *txL* e os sinais *sizeSend* e *data_outL* recebem *Mdata_i* (que corresponde ao tamanho do *payload* do pacote) + ‘1’ (por causa do comando OCP que deve ser inserido no pacote) (*flit* 2). Quando o sinal *ack_txL* é ativo passa-se para o próximo estado.

SS4: neste estado, o módulo IR escravo-OCP recebe o primeiro dado do *payload.* No mesmo estado, o sinal *auxScmdAccept* é ativo e o sinal *txL* é desativado. Quando o sinal *Mcmd_i* receber um comando válido e o sinal *auxScmdAccept* for desativado passa-se para o estado SS5.

SS5: o sinal *txL* é ativo. No mesmo estado, o sinal *data_outL* recebe o comando OCP do sinal *Mcmd_i* (*flit* 3). O contador de *flit*s é incrementado. Quando o sinal *ack_txL* é ativo passa-se para o próximo estado.

SS6: o sinal *txL* é desativado. No momento que o sinal *ack_txL* é ativo passa-se para o próximo estado.

SS7: no último estado da máquina ativa-se o sinal *txL*. O sinal *data_outL* recebe o conteúdo do próximo *flit*. Quando não houver mais *flits* a transmitir, retorna-se ao estado IDLE.

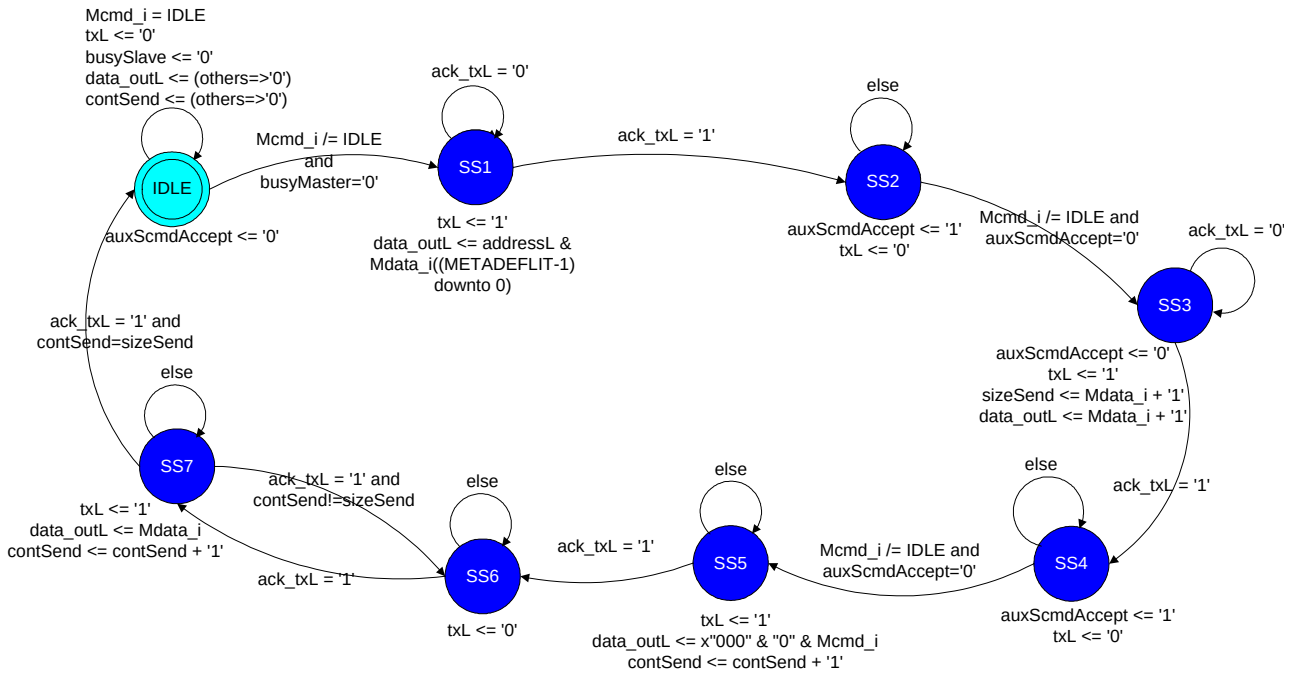


Figura 38 - Máquina de estados do módulo IR-E responsável pela recepção das requisições OCP providas pelo núcleo com IR-M.

A **máquina de recepção *chave*** recebe respostas (pela porta local da chave) às leituras previamente requisitas pelo núcleo mestre, montando os pacotes de leitura. Estes pacotes devem ser enviados a interface de rede mestre-OCP a partir das transações descritas na Tabela 7. Observa-se na Figura 39, que no primeiro estado descartam-se os endereços da chave fonte e destino, ou seja, apenas os dados de resposta são enviados ao núcleo com IR-M.

A descrição dos estados que compõem esta máquina é apresentada abaixo:

IDLE: neste estado, todos os sinais de saída deste módulo são inicializados em ZERO. Ao detectar que o sinal *rxL* é ativo passa-se para o estado RS1.

RS1: neste estado, o conteúdo do sinal *data_inL* (primeiro *flit*) é desconsiderado, pois este contém a informação do endereço da chave fonte e do destino, informação esta não necessária para o núcleo mestre. Quando *rxL* for ativo passa-se para o próximo estado.

RS2: no presente estado, armazena-se o tamanho do *payload*, ou seja, o número de *flits* que devem ser enviados ao núcleo.

RS3: o sinal *cmdOCP* recebe o conteúdo do sinal *data_inL*, o qual corresponde ao comando OCP. No mesmo instante, o sinal *Sresp_o* recebe um valor NULO. Estabeleceu-se que quando uma resposta à leitura trafega pela rede, o comando OCP associado contém o valor *9* (inválido para OCP). Desta forma é possível separar o tráfego de requisições de leitura/escrita do tráfego de respostas à leitura. Caso o tráfego proveniente da rede contiver o comando *9*, a máquina interpreta isto como uma resposta à leitura indo para o estado RS4. Caso o comando for diferente de *9*, isto significa um pacote direcionado a

uma porta escravo, a qual não pode por definição receber este tipo de tráfego. Assim, estes pacotes são descartados, no laço compreendido entre os estados RS5 e RS6.

RS4: neste estado, é informado ao mestre-OCP que o primeiro dado do *payload* está disponível no sinal *Sdata_o*, sinaliza-se dado disponível (*Sresp_o* ← DVA) e incrementa-se o contador de *flits* enviados (cont + 1).

RS5: incrementa-se o contador de *flits* e quando *rxL* for ativo passa-se para o próximo estado (*flit* descartado, não enviado para o mestre).

RS6: enquanto todos os *flits* não forem enviados (cont != size), retorna-se ao estado anterior.

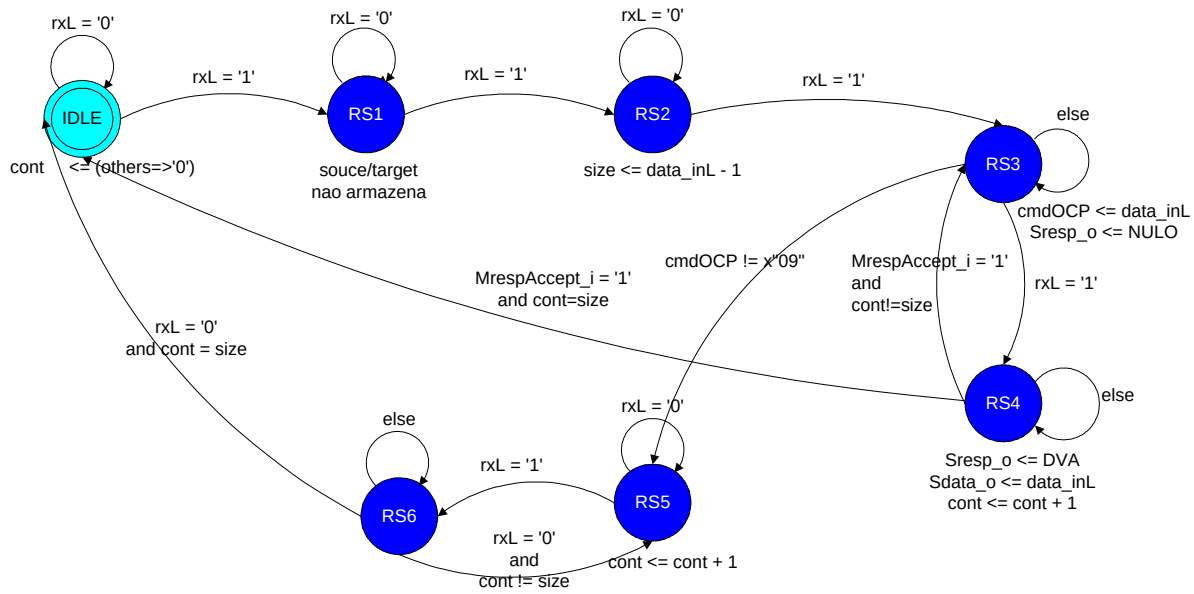


Figura 39 - Máquina de estados do módulo IR-E responsável pela recepção e montagem dos pacotes provindos da rede.

4.2.2.1 Validação funcional da IR-E OCP

O comportamento das máquinas descritas acima foram validados por simulação funcional, conforme ilustram a Figura 40 (máquina de estados da Figura 38 e Figura 39). A Figura 40 apresenta o envio de dois pacotes, um enviado a partir de transações de escrita e o outro com transações leitura (os dois primeiros *flits* da operação de leitura são transações de escrita, correspondendo ao endereço destino e tamanho do *payload*).

A descrição dos itens numerados na Figura 40 é apresentada abaixo:

1. A fase de requisição de escrita inicia quando no sinal *Mcmd_i* recebe WR (valor 1). No mesmo instante, os sinais *Madd_i* e *Mdata_i* recebem “0” e o dado a ser escrito respectivamente. Deve-se salientar que o sinal *Madd_i* não esta sendo utilizado porque o endereço de destino (“0011”) é enviado pelo sinal *Mdata_i.*

2. Um ciclo de relógio depois da requisição de escrita, o sinal *txL* é ativo, informando a presença de *flits* a serem transmitidos. No mesmo ciclo de relógio o sinal *ack_txL* é ativo, permitindo colocar o *flit* no sinal *data_outL*.

3. A chave detecta que o sinal *rx*(4) da porta local (índice 4 conforme a Figura 29 da página 42) esta ativo. No mesmo instante o sinal *data_in*(4) recebe o conteúdo do *flit*. Por conseguinte, o *flit* é armazenado na fila da porta local.

4. Um ciclo de relógio depois o sinal *ScmdAccept_o* recebe o sinal *auxScmdAccept*, que está ativo, finalizando a fase de escrita. Cada *flit* precisa de 4 ciclos de relógio para a execução desse protocolo, sendo 2 ciclos para o protocolo OCP e 2 para o protocolo da rede.

5. O sinal *Mcmd_i* recebe RD (valor 2), iniciando a fase de requisição de leitura. Neste caso, cada *flit* corresponde ao endereço de leitura do dado desejado. Deve-se ressaltar que os dois *flits* (“0011” e “0006”) anteriores a fase de requisição de leitura contém, respectivamente, o destino e o número de dados que devem ser retornados ao solicitante das leituras. Observar que para primeira transação de leitura são necessários 7 ciclos de relógio (necessário para adição do comando OCP no pacote), sendo que para as demais utilizam-se 4 ciclos.

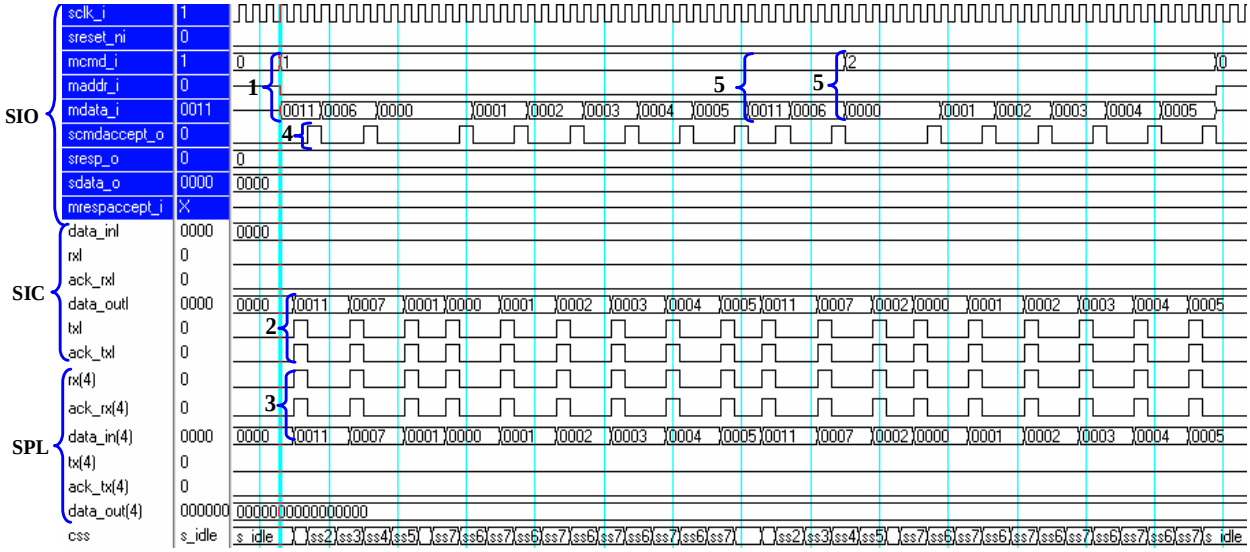
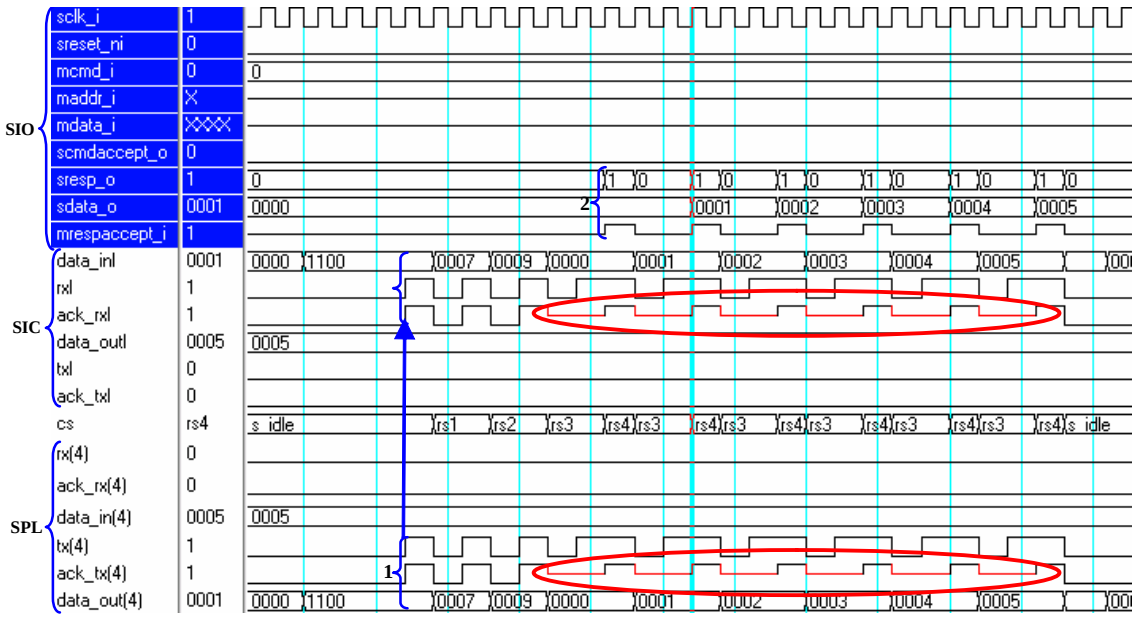


Figura 40 - Simulação da máquina responsável pela recepção de requisições OCP e montagem do pacote para envio na rede (porta local da chave). SIO corresponde aos sinais da interface OCP, SIC aos sinais da interface IR-E com a chave (porta local) e SPL refere-se aos sinais da porta local da chave.

Figura 41 é apresentada abaixo:

1. A chave coloca o *flit* no sinal *data_out*(4) e ativa o sinal *tx*(4) da porta local. Uma vez que o sinal *ack_tx*(4) está ativo o *flit* é removido do buffer e capturado do sinal *data-inL*. O terceiro *flit* capturado do sinal *data_inL*, com valor *0009*, corresponde ao comando OCP inválido, descrito anteriormente, que indica uma ou mais respostas do escravo a uma ou mais solicitações do núcleo mestre conectado a chave.

2. O sinal *Sresp_o* informa que existe um dado disponível no sinal *Sdata_o*. A cada resposta valida (*Sresp_o* = “1”) o sinal *MrespAccept_i* é ativo pelo núcleo conectado à chave e o dado é capturado pelo mesmo.



Observar, que os sinais com valor indeterminado (XXXX) são gerados pelo módulo QC-Slave, responsável pelo protocolo de envio/recepção de dados.

Figura 41 - Simulação da máquina responsável pela recepção e montagem dos pacotes provindos da rede para a IR-M do núcleo. SIO corresponde aos sinais da interface OCP, SIC aos sinais da interface IR-E com a chave (porta local) e SPL refere-se aos sinais da porta local da chave.

4.2.3 Interface de rede mestre OCP

As interfaces de rede na NoC do tipo mestre permitem a conexão de núcleos com interface de rede do tipo escravo (ver na Figura 34 os módulos *qcslave0, qcslave1* e *qcslave2* conectados às IRs do tipo mestre M_*ocp_10*, M_*ocp_11 e* M_*ocp_01* , respectivamente). Os sinais OCP que compõem uma IR-M são descritos na Tabela 9.

Tabela 9 - Sinais OCP da Interface Mestre.

Grupo	Sinal	Largura(bits)	Driver
Básico	MClk_i	1	Mestre / Escravo
	MAddr_o	1	Mestre
	MCmd_o	3	Mestre
	MData_o	8, 16, 32 ou 64	Mestre
	SCmdAccept_i	1	Escravo
	SData_i	8, 16, 32 ou 64	Escravo
	SResp_i	2	Escravo
Fluxo de dados	MRespAccept_o	1	Mestre
Opcionais de controle	MReset_ni	1	Mestre / Escravo

A lógica responsável pela segmentação e remontagem de pacotes foi dividida em duas máquinas de estado: (*i*) **máquina de recepção *chave* -** recepção e envio dos *flits* provindos da rede através da geração de transações OCP para a IR-E do núcleo; e (*ii*) **máquina de recepção *núcleo* -** que a partir das respostas providas do núcleo com IR-E monta os pacotes para enviar pela rede ao solicitante das mesmas.

A **máquina de recepção *chave*** recebe pacotes da rede (porta local da chave) referentes à

requisições de leitura ou de escrita de dados ao/no núcleo com IR-E. Estes pacotes devem ser enviados a interface de rede escravo-OCP a partir das transações descritas nas Tabela 5 e Tabela 6. O comportamento da IR-M responsável pela recepção e envio dos *flits* provindos da rede através da geração de transações OCP para a IR-E do núcleo é ilustrado na Figura 42.

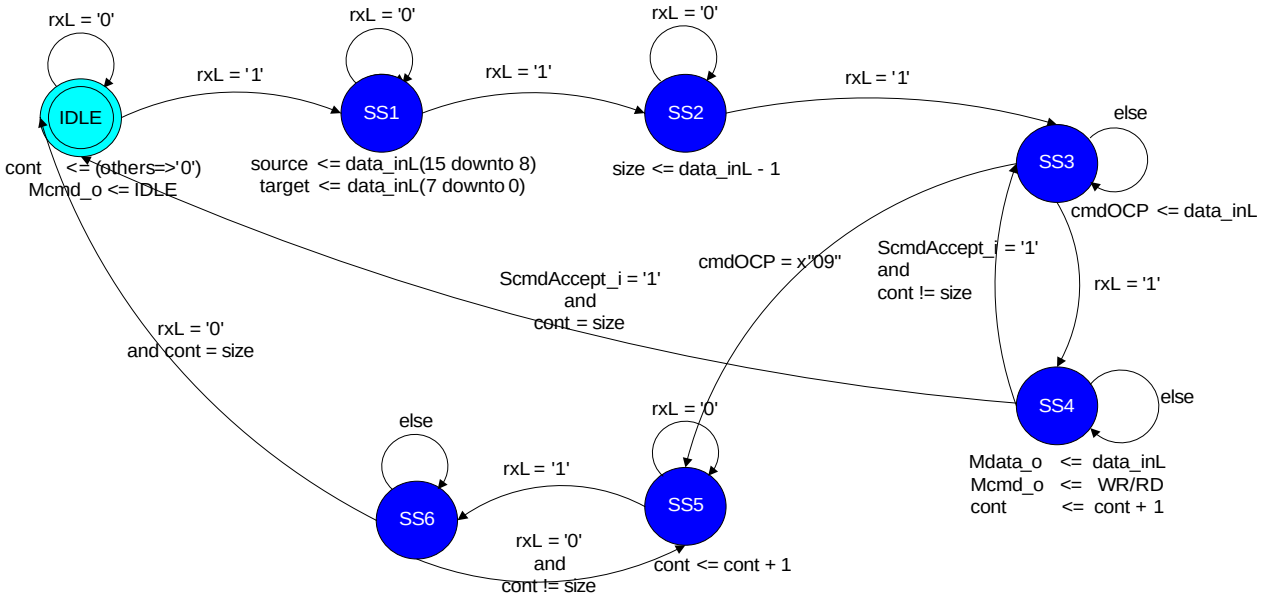


Figura 42 - Máquina de estados do módulo IR-M responsável pela recepção de pacotes provindo pela rede e envio dos *flits* através de transações OCP.

A descrição dos estados que compõem a máquina do módulo IR-M, Figura 42, segue:

IDLE: Uma vez que o sinal *rxL* for ativo (chegou dado da rede na porta local da chave) passa-se para o próximo estado.

SS1: No presente estado, os sinais *source* e *target* recebem os endereços fonte e destino, respectivamente. Quando o sinal r*xL* é ativo passa-se para o próximo estado.

SS2: O sinal *size* recebe o *flit* que contém o tamanho do *payload* a ser enviado. Enquanto o sinal r*xL* estiver desativado, permanece-se neste estado.

SS3: Neste estado, captura-se o *flit* que contém o comando OCP do sinal *data_inL*. Se o comando OCP associado à esse *flit* contém o valor *9*, a máquina interpreta isto como uma resposta à leitura indo para o estado SS5. Isto significa que este pacote deve ser excluído, pois por definição um núcleo com IR-E não pode inicializar operações e, por conseqüente, não pode receber este tipo de tráfego. Sendo assim, estes pacotes são descartados, no laço compreendido entre os estados SS5 e SS6. Caso o comando for igual a WR/RD passa-se para o estado SS4.

SS4: Neste estado, inicializa-se a fase de requisição a partir do comando OCP contido no sinal *cmdOCP*, ou seja, atribuindo-se WR ou RD ao sinal *Mcmd_o*. No mesmo instante, o sinal *Mdata_o* recebe o primeiro *flit* a ser transferido e incrementa-se o contador (cont + 1). Fica-se nesse laço, estados SS3 e SS4, até que todos os *flits* do payload tenham sido enviados.

SS5: O contador de *flit*s é incrementado. Quando o sinal r*xL* é ativo passa-se para o estado SS6 (*flit* descartado, não enviado para o núcleo).

SS6: No último estado dessa máquina, o contador é incrementado até que o número de *flits* enviado for igual ao tamanho do *payload* recebido no estado SS2.

Na **máquina de recepção *núcleo*** observa-se que para cada *flit* recebido pela IR-M são necessários dois estados: um para capturar e informar que existe um *flit* a ser transmitido (*txL* = "1") e outro para receber a confirmação que o *flit* foi transmitido à porta local da chave. O comportamento da IR-M OCP responsável pela montagem e envio de pacotes para rede é ilustrado na Figura 43. Esta máquina corresponde às respostas de pedidos de leitura por algum núcleo mestre conectado à rede.

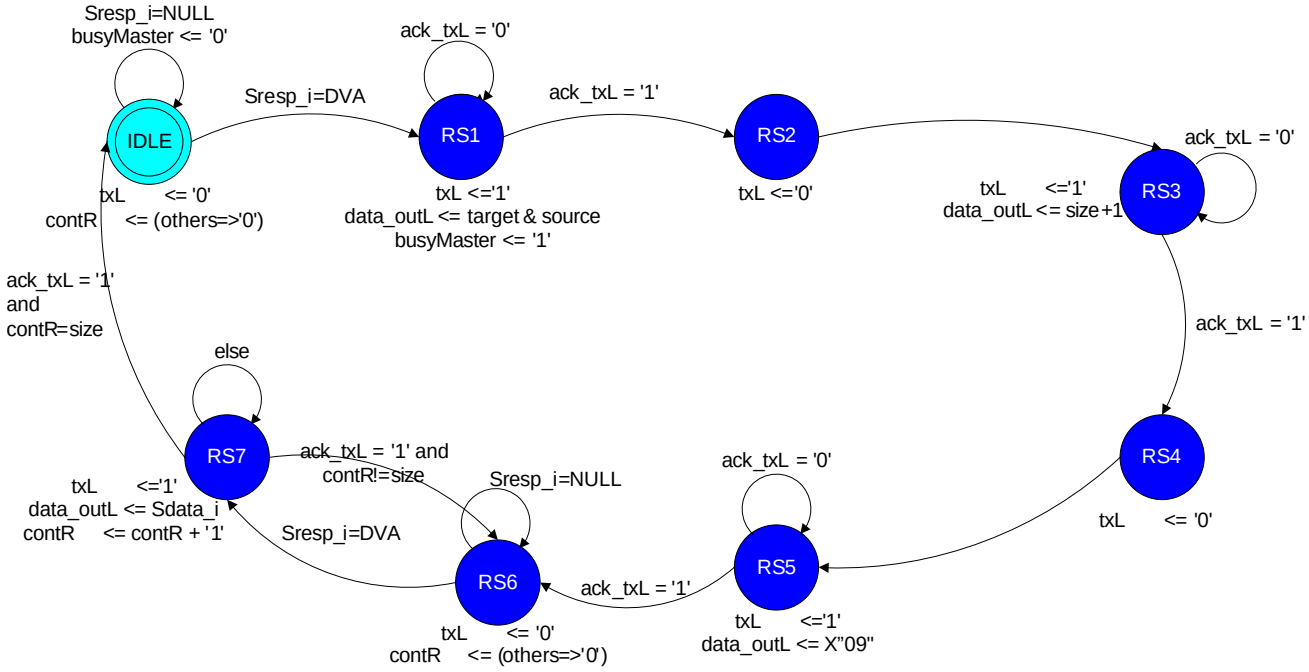


Figura 43 - Máquina de estados de recepção *núcleo* do módulo IR-M OCP.

A descrição dos estados que compõem esta máquina é apresentada abaixo:

IDLE: Ao receber uma resposta válida (DVA) no sinal *Sresp_i* e se o sinal *busyMaster* não estiver ativo passa-se para o estado SS1. Uma explicação mais detalhada referente a utilidade do sinal *busyMaster* é apresentada Seção 4.2.4 desse Capítulo.

RS1: Neste estado, recebe-se o primeiro *flit* cujo conteúdo corresponde ao endereço de destino (solicitando dos dados) e o endereço do fonte (origem dos dados). Este *flit* é enviado à rede através do sinal *data_outL*. No mesmo instante, ativa-se os sinais *txL* e *busyMaster*. Quando o sinal *ack_txL* for ativo passa-se para o próximo estado.

RS2: No presente estado, o sinal *txL* é desativado.

RS3: O sinal *txL* é ativo e sinal *data_inL* recebe o segundo *flit* que contém o tamanho do *payload*. Quando o sinal *ack_trxL* for ativo passa-se para o próximo estado.

RS4: Neste estado, o sinal *txL* é desativado.

RS5: Neste estado, o sinal *txL* é ativo e sinal *data_inL* recebe um comando interno (“09”) utilizado para identificar uma resposta de leitura.

RS6: O sinal *txL* é desativado.

RS7: No último estado da máquina ativa-se o sinal *txL*. O sinal *data_outL* recebe o conteúdo do próximo *flit*. No mesmo instante, o contador é incrementado até o envio de todos os *flits*.

4.2.3.1 Validação funcional da IR-M OCP

Os comportamentos das máquinas descritas acima foram validados por simulação funcional, conforme ilustra a Figura 44.

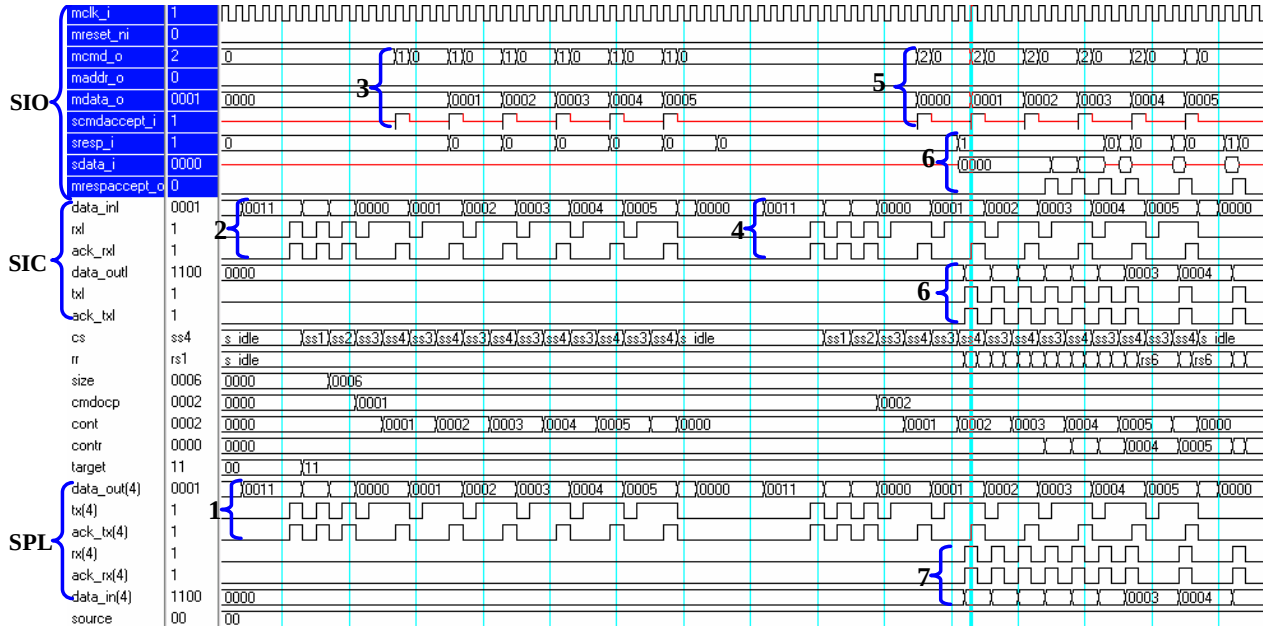


Figura 44 - Simulação das máquinas de estado de recepção *núcleo* e de recepção *chave* da IR-M. SIO corresponde aos sinais da interface OCP, SIC aos sinais da interface IR-M com a chave (porta local) e SPL refere-se aos sinais da porta local da chave.

A descrição dos itens numerados na Figura 44 é apresentada abaixo:

1. A chave 11 coloca o *flit* no sinal *data_out*(4) e ativa o sinal *tx*(4) (porta local). O primeiro *flit* corresponde aos endereços da chave fonte e destino, respectivamente.

2. O módulo IR-M detecta que o sinal r*xL* é ativo, informando a presença de *flits* a serem recebidos. No mesmo ciclo de relógio o sinal *ack_txL* é ativo, permitindo capturar o *flit* no sinal *data_inL* e a conexão entre a chave e o módulo IR-M é estabelecida.

3. O módulo IR-M inicia uma transação de escrita OCP, *Mcmd_o* = ‘1’ (observar que os dois primeiros *flits* foram descartados). No mesmo instante, o sinal *Mdata_o* recebe o *flit* que deve ser enviado. O módulo *QC-Slave* aceita a transferência no mesmo clico de relógio (*Scmdaccept_i* = “1”), permitindo assim que o módulo IR-M envie o *flit*. Este processo ocorre até o envio de todos os *flits*.

4. O módulo IR-M recebe um segundo pacote cujo terceiro *flit* indica o tipo de transação a ser efetuada, no caso transação de leitura. Os demais *flits* referem-se aos endereços dos dados a serem lidos.

5. O módulo IR-M inicia uma transação de leitura OCP (*Mcmd_o* = ”2”). O módulo *QC-Slave* aceita a transferência no mesmo ciclo de relógio (*Scmdaccept_i* = “1”), permitindo assim que o módulo IR-M envie o *flit* com o endereço do dado a ser lido.

6. Após 3 ciclos de relógio o módulo *QC-Slave* apresenta uma resposta válida no sinal *Sresp_i* (“1”). Depois de 6 ciclos de relógio, o módulo IR-M está pronto para aceitar a resposta, ou seja, capturar o *flit* do sinal *Sdata_i* no sinal *data_outL*. Este processo ocorre até que os dados desejados sejam lidos do módulo *QC-Slave* e enviadas pela rede até o solicitante dos mesmos.

7. Neste instante, a chave detecta que o sinal *rx*(4) da porta local esta ativo. No mesmo instante o sinal *data_in*(4) recebe o conteúdo do *flit*. Por conseguinte, o *flit* é armazenado na fila de entrada da porta local até o estabelecimento de conexão com uma das portas de saída.

4.2.4 Interface de rede mestre-escravo OCP

A IR-ME é um invólucro que envolve tanto o módulo mestre (IR-M) como o módulo escravo (IR-E). Sendo assim, destacam-se duas situações que devem ser consideradas no desenvolvimento do mesmo: (*i*) disputa entre os módulos pela porta local para envio de pacotes; e (*ii*) direcionamento dos pacotes provindos da rede:

(i) em um mesmo momento ambos os módulos podem desejar utilizar a porta local da chave para envio de pacote(s), tanto o mestre querendo enviar pacotes (transação de escrita/leitura) quanto o escravo querendo responder a uma leitura.

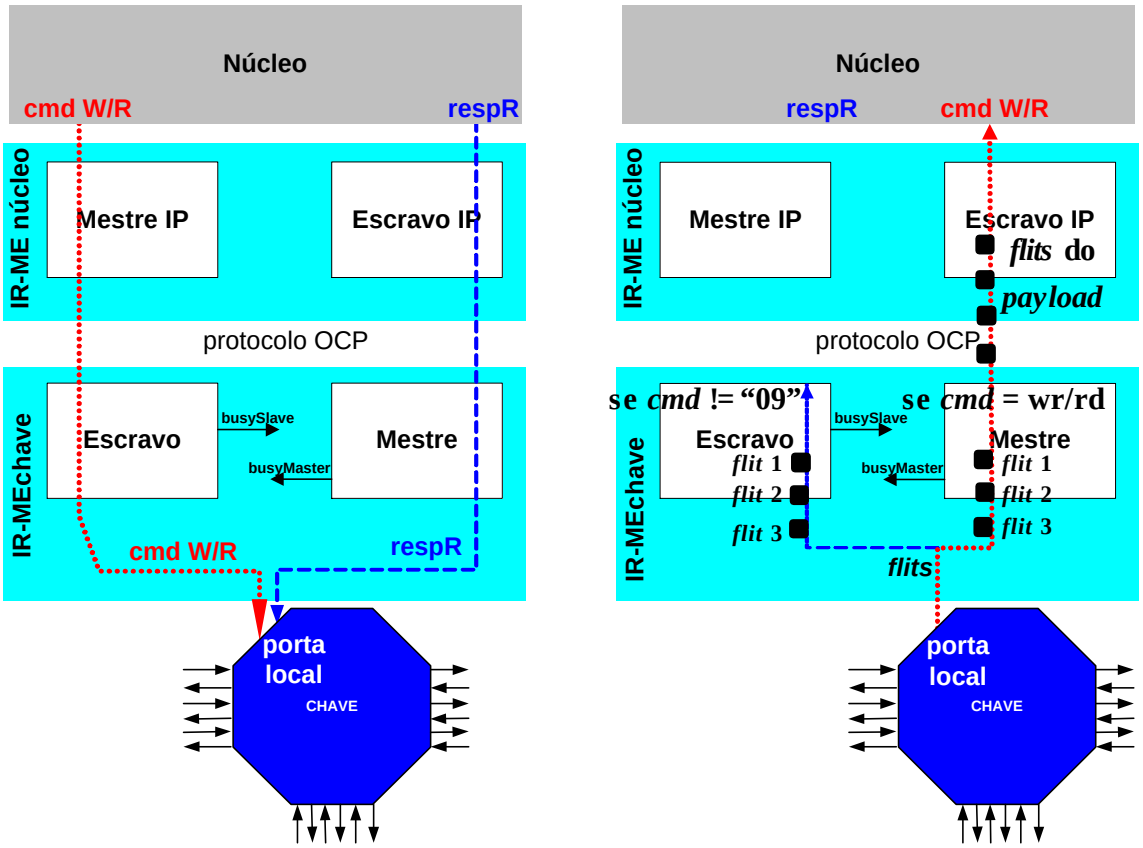
(ii) ao receber um pacote enviado pela rede deve-se diferenciar se este pacote é direcionado para o módulo mestre (resposta a uma leitura) ou para o módulo escravo (comando de escrita/leitura).

Para controlar a disputa entre os módulos mestre e escravo à porta local utiliza-se um sinal que deve ser ativo toda a vez que um dos módulos tiver acesso à rede (porta local). Este sinal é denominado de *busyMaster* para módulo mestre e de *busySlave* para o módulo escravo. Quando um destes módulos quiser enviar dados para a rede deve-se verificar se o sinal de *busy* do vizinho está ativo. Se o vizinho estiver com sinal *busy* em zero, o *busy* do mesmo deve ser ativado e mantido em nível lógico 1 até o envio de todos os *flits*. Para garantir o funcionamento dessa implementação, no módulo escravo este sinal é testado na *borda de subida* do relógio, enquanto no mestre o teste se dá na *borda de decida* do relógio.

Para contornar a situação (*ii*), é enviado no terceiro *flit* um comando OCP válido indicando que o pacote deve ser entregue ao mestre. Caso contrário, o terceiro *flit* recebe um comando OCP inválido, direcionando o mesmo ao escravo. A recepção deste comando inválido pode ser observado

pela interpretação do comando ‘*9*’, nas Figura 39 (estados RS3-RS4) e Figura 42 (estados SS3-SS4).

A Figura 45 apresenta as duas situações descritas acima. Na situação 1 (Figura 45(a)), chegam simultaneamente uma requisição de escrita/leitura (cmd W/R) provinda do *Mestre IP* ao módulo *Escravo* (IR-ME da chave) e uma resposta (respR) enviada pelo *Escravo IP* ao módulo *Mestre*. Sendo assim, os módulos *Mestre* e *Escravo* disputam acesso à porta local da chave. A Figura 46 apresenta a simulação funcional da solução implementada para contornar o fenômeno descrito acima (situação 1).



(a) situação 1 – requisições de escrita (W)/ leitura (R) do módulo *Mestre-IP* chegando à porta local da chave no mesmo instante que uma resposta provinda do *Escravo-IP*.

(b) situação 2 – recepção de *flits* provindos da rede (porta local) e envio destes aos dois módulos (*Mestre* e *Escravo*).

Figura 45 - Funcionalidade de um módulo IR-ME.

Já na situação 2, pacotes contendo respostas/requisições são recebidos pela porta local e divididos em *flits*. Os três primeiros *flits* do pacote são enviados aos módulos *Mestre* e *Escravo* da IR-ME da chave (Figura 45 (b)). Ao receber o terceiro *flit* cada módulo verifica o comando contido no mesmo, se esse comando for válido os *flits* restantes são enviados ao seu destino na IR-ME do núcleo. Por exemplo, se o comando (*cmd*) do *flit* 3 for igual à WR/RD (como ilustrado na Figura 45 (b)) os demais *flits* (*flits* do *payload*) são enviados ao *Escravo IP* que está conectado ao *Mestre* da IR-ME da chave. Por sua vez, o módulo *Escravo* descarta os demais *flits*.

4.2.4.1 Validação funcional da IR-ME OCP

A Figura 46 apresenta a validação funcional da IR-ME OCP. A descrição dos itens numerados na Figura 46 é apresentada abaixo.

1. O módulo *Mestre* recebe uma resposta válida no sinal *Sresp_i* (“1”) e o dado de resposta no sinal *Sdata_i* (“1234”). No mesmo ciclo de relógio, o módulo *Escravo* recebe uma requisição de escrita (*Mcmd_i* = WR) provinda do *Mestre IP* da IR-ME do núcleo. Nesse instante, o módulo *Escravo* ativa o sinal *busySlave* informando ao *Mestre* que a porta local esta sendo utilizada. Por sua vez, o módulo mestre fica esperando a desativação do sinal *busySlave*, ou seja, a liberação da porta local.

2. Tendo acesso à porta local, o *Escravo* ativa o sinal *txL*, informando a presença de *flits* a serem transmitidos. O primeiro dado é colocado no sinal *data_outL* (“0001”) e após um ciclo de relógio, o sinal *ack_txL* é ativo, permitindo o envio do mesmo a rede (porta local).

3. Após enviar todos os *flits*, o módulo *Escravo* desativa o sinal *busySlave* permitindo assim, que *Mestre* tenha acesso à porta local da chave.

4. No mesmo ciclo de relógio, o sinal *txL* é ativo. Após um ciclo de relógio, o primeiro dado é enviado a rede (*ack_txL* = “1”).

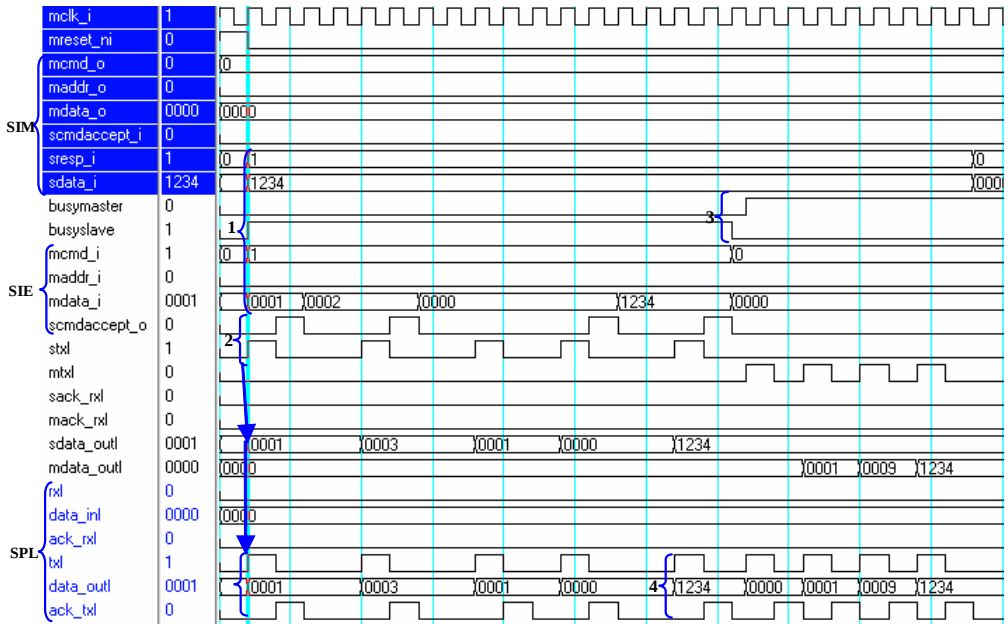


Figura 46 - Simulação da disputa entre o módulo *Mestre* e *Escravo* da IR-ME pela porta local da chave. SIM indica os sinais do *Mestre*, SIE os sinais do módulo *Escravo* e SPL os sinais da porta local.

4.3 Certificação das IRs

Como descrito anteriormente, as IRs OCP desenvolvidas no presente trabalho foram validadas através da ferramenta *CoreCreator*. Depois de executar as fases descritas na Seção 3.2.4 (página 33), a ferramenta *CoreCreator* gera arquivos que representam as transações OCP ocorridas entre duas entidades. No contexto desse trabalho, entre uma chave da rede e um módulo QC-Master/Slave.

Para facilitar a compreensão do relatório ilustrado na Figura 47 segue o significado de cada sigla utilizada no relatório: (*i*) *SimTime*- tempo de simulação; (*ii*) *Cycle* - tempo de ciclo válido; (*iii*) CAD (*Command Accept Delay*) - número de ciclos de relógios entre a solicitação de uma transação

e o aceite dessa pelo escravo; (iv) RVD (*Response Valid Delay*) - número de ciclos de relógios entre uma solicitação de leitura e apresentação de um dado válido (Resp = DVA); (v) RAD (*Response Accept Delay*) - número de ciclos de relógios entre uma solicitação de leitura e um dado é aceito (*MRespAccept* é ativo). O relatório da Figura 47 prove alguns dos sinais e os valores envolvidos nas transações ocorridas para o envio/recepção de pacotes do módulo *QC-Master* (conectado a chave 00) para/do (o) módulo *QC-Slave* (conectado a chave 11), conforme simulações ilustradas nas Figura 40 e Figura 41.

	SimTime	Cycle	CAD	Cmd	Addr	Data	Resp	RVD	RAD
	2750	27	2	WRITE	0	0011		.	.
	3050	30	3	WRITE	0	0006		.	.
	3450	34	6	WRITE	0	0000		.	.
A	4150	41	3	WRITE	0	0001		.	.
	4550	45	3	WRITE	0	0002		.	.
	4950	49	3	WRITE	0	0003		.	.
	5350	53	3	WRITE	0	0004		.	.
	5750	57	3	WRITE	0	0005		.	.
	6150	61	2	WRITE	0	0011		.	.
	6450	64	3	WRITE	0	0006		.	.
	6850	68	6	READ	0	0000	DVA	83	83
	7550	**75**	**3**	**READ**	**0**	**0001**	**DVA**	**79**	**79**
	7950	79	3	READ	0	0002	DVA	78	78
	8350	83	3	READ	0	0003	DVA	77	77
	8750	87	3	READ	0	0004	DVA	76	76
	9150	91	3	READ	0	0005	DVA	75	75

No item A Figura 47 estão representadas as transações de escrita referentes ao envio do primeiro pacote (descrito na Figura 35). Por exemplo, no tempo de simulação 2750 ns, inicia-se a fase de requisição para uma transação de escrita (data = "0011", cujo valor corresponde ao endereço da chave destino). Observar no campo CAD ("2"), que após dois ciclos de relógio o lado escravo OCP (IR-E da chave) aceita a transação de escrita. A partir das informações apresentadas no relatório acima observa-se que para a primeira transação de leitura, desde a solicitação até a recepção de uma resposta válida DVA, foram necessários 6 ciclos OCP, sendo que para as demais transações são necessários apenas 3 ciclos OCP (e.g. transação em destaque, ciclo OCP 75). Como descrito anteriormente, estes 3 ciclos de relógio são referentes a adição do comando OCP no pacote que deve ser enviado ao seu destino, conforme ilustrado na Figura 40.

Figura 47 - Relatório gerado a partir do tráfego STL especificado nas Figura 35 e Figura 36. Estes valores são provenientes do monitor *ocp0* utilizado para verificar as transações OCP entre a IR-E da chave 00 e o módulo *QC-Master* (Figura 34).

O relatório II ilustrado na Figura 48, diferencia-se do relatório anterior pois este é mais detalhado em função da representação de todos os sinais e valores envolvidos nas transações OCP durante cada ciclo. O item A representado na Figura 48 contém todas as transações de escritas ocorridas. No relatório anterior (Figura 47), o inicio da primeira transação de escrita ocorre as 2750 ns, já no relatório II esta transação é aparece aos 2950 ns, dois ciclos de relógio referentes a latência de aceite do escravo (sinal *ScmdAccept*, representado no relatório II pela letra A). No item B, pode-se observar o inicio das transações de leitura, onde o escravo OCP (IR-E chave00) não aceitou a transação e os valores de resposta são inválidos (*Resp*= "NULL"). Por exemplo, na 75ª transação o módulo *QC-Master* solicita uma leitura, porém a resposta de aceite (sinal *ScmdAccept* ativo) ocorre 3 ciclos de relógio depois (78ª transação). Por fim, o item C apresenta a recepção de valores validos e a finalização das transações de leitura (sinal *MRespAccept*, representado na Figura 48 por M). Além dos relatórios descritos acima, salienta-se o gerado pelo módulo *ocpcheck*. Este não é apresentado dado a semelhança ao ilustrado na Figura 27.

```
                Time      Cycle  R Cmd       A Addr MData Resp M SData
```

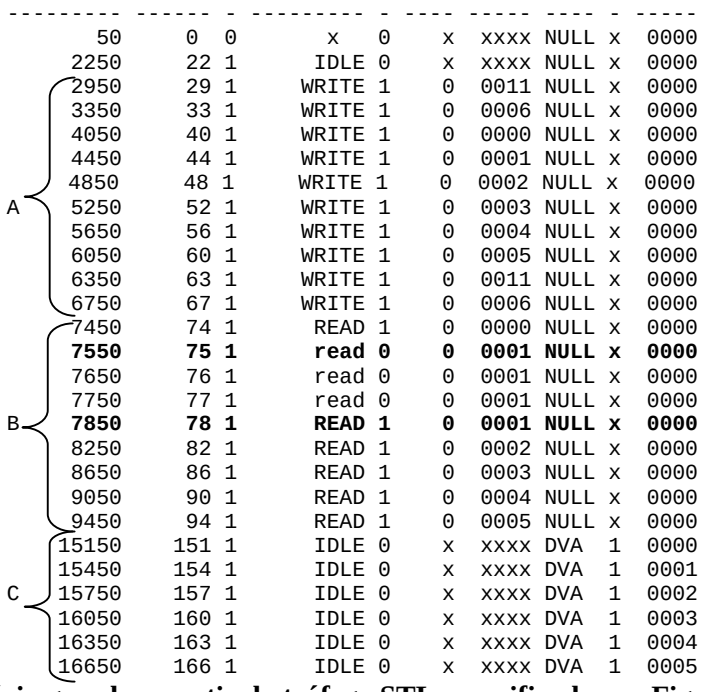


Figura 48 - Relatório gerado a partir do tráfego STL especificado nas Figura 35 e Figura 36.

4.4 Considerações Finais do Capítulo

A IRs OCP desenvolvidas implementam um mecanismo de controle de fluxo de dados baseado em *handshake*. Ou seja, é permitido o envio de *flits* apenas se o receptor tiver capacidade de absorvê-los o que evita a possibilidade de perda de pacotes.

As interfaces de rede descritas nesse Capítulo permitem que várias transações sejam enviadas por um mestre a um ou mais escravo(s) mesmo antes do recebimento do primeiro pacote de resposta. Dessa forma pacotes de resposta podem ser recebidos em uma ordem diferente daquela na qual os pacotes de requisição foram enviados. Uma forma de contornar essa deficiência é a utilização de filas nas IRs que devem ser utilizadas para garantir a entrega das respostas ao núcleo na mesma ordem das requisições enviadas pelo mesmo.

Como trabalho futuro cita-se também a expansão do grupo de sinais das interfaces de rede OCP. Objetiva-se com isso, suportar a transferência de dados utilizando o modo rajada e *threads*, conforme especificado em [OCP03].

5 FERRAMENTA PARA PARAMETRIZAÇÃO DA REDE HERMES

A parametrização das chaves e a geração manual da rede a partir destas é um processo passível de erros, devido à grande quantidade de fios que ligam as chaves entre si e ao núcleo local, como pode ser observado na Figura 49. Tipicamente, cada chave tem 30 canais de entrada/saída. A automatização deste processo foi uma das motivações para o desenvolvimento da ferramenta Maia, segunda contribuição deste trabalho.

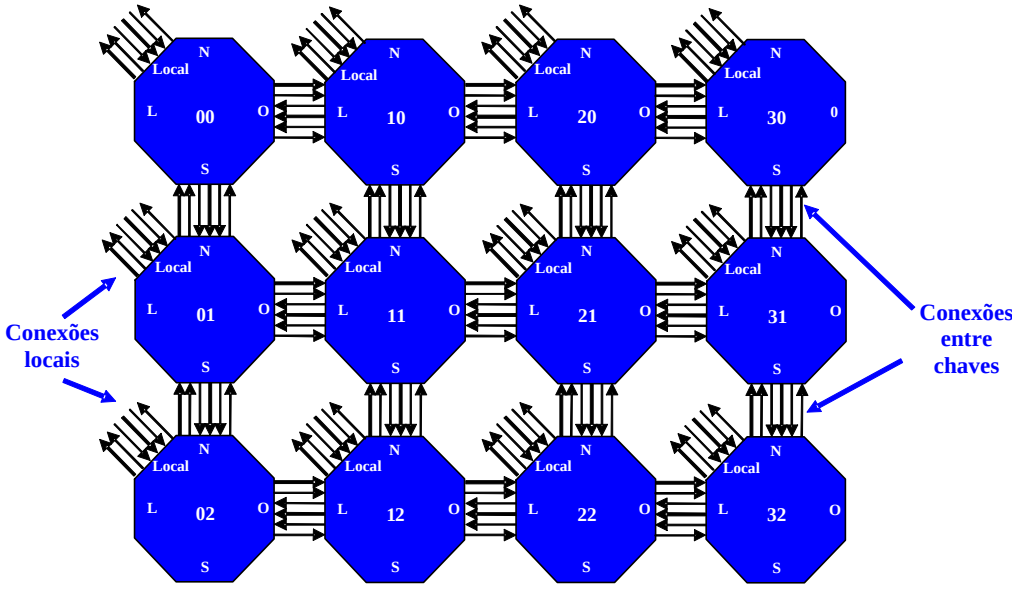


Figura 49 - Interconexões entre chaves em uma rede intra-chip 3 x 4.

A Maia foi desenvolvida em linguagem JAVA e seu o desenvolvimento inicial (parametrização do código VHDL) ocorreu em cooperação entre o autor desta dissertação e o doutorando José Palma (UFRGS). Atualmente trabalham na ferramenta o autor desse trabalho, e a bolsista Aline Vieira, que atua no grupo GAPH.

Além de automatizar o processo de interconexão das chaves, a ferramenta permite a escolha do algoritmo de roteamento e a configuração de parâmetros como: (*i*) profundidade das filas; (*ii*) largura do *flit*; (*iii*) dimensão da rede. Pretende-se a partir da configuração desses parâmetros gerar redes intra-chip que respeitem os requisitos de uma aplicação especifica.

Uma outra funcionalidade da ferramenta é a remoção de filas de entrada de canais não utilizados na chave, o que reduz a área da HERMES. Além disso, a ferramenta gera tráfego e *testbenchs*, permitindo com isso validar a rede gerada. A descrição da parametrização do código da HERMES e as funcionalidades suportadas pela ferramenta são descritas a seguir.

5.1 Parâmetros Arquiteturais

Para gerar uma NoC, a ferramenta Maia utiliza arquivos VHDL que descrevem a rede HERMES. A partir destes arquivos, a ferramenta gera a estrutura da NoC desejada em função dos parâmetros definidos pelo usuário. Para que os arquivos VHDL da rede HERMES possam ser utilizados pela ferramenta, foram inseridos *flags* que definem os pontos do código a serem alterados pela mesma.

5.1.1 Parametrização da biblioteca *Hermes_package*

Dentre os arquivos alterados destaca-se o *Hermes_Package.vhd* (biblioteca específica da rede HERMES). Nesta biblioteca são inseridos os valores configurados pelo usuário. A Figura 50 apresenta um trecho do código VHDL da biblioteca *Hermes_Package*.

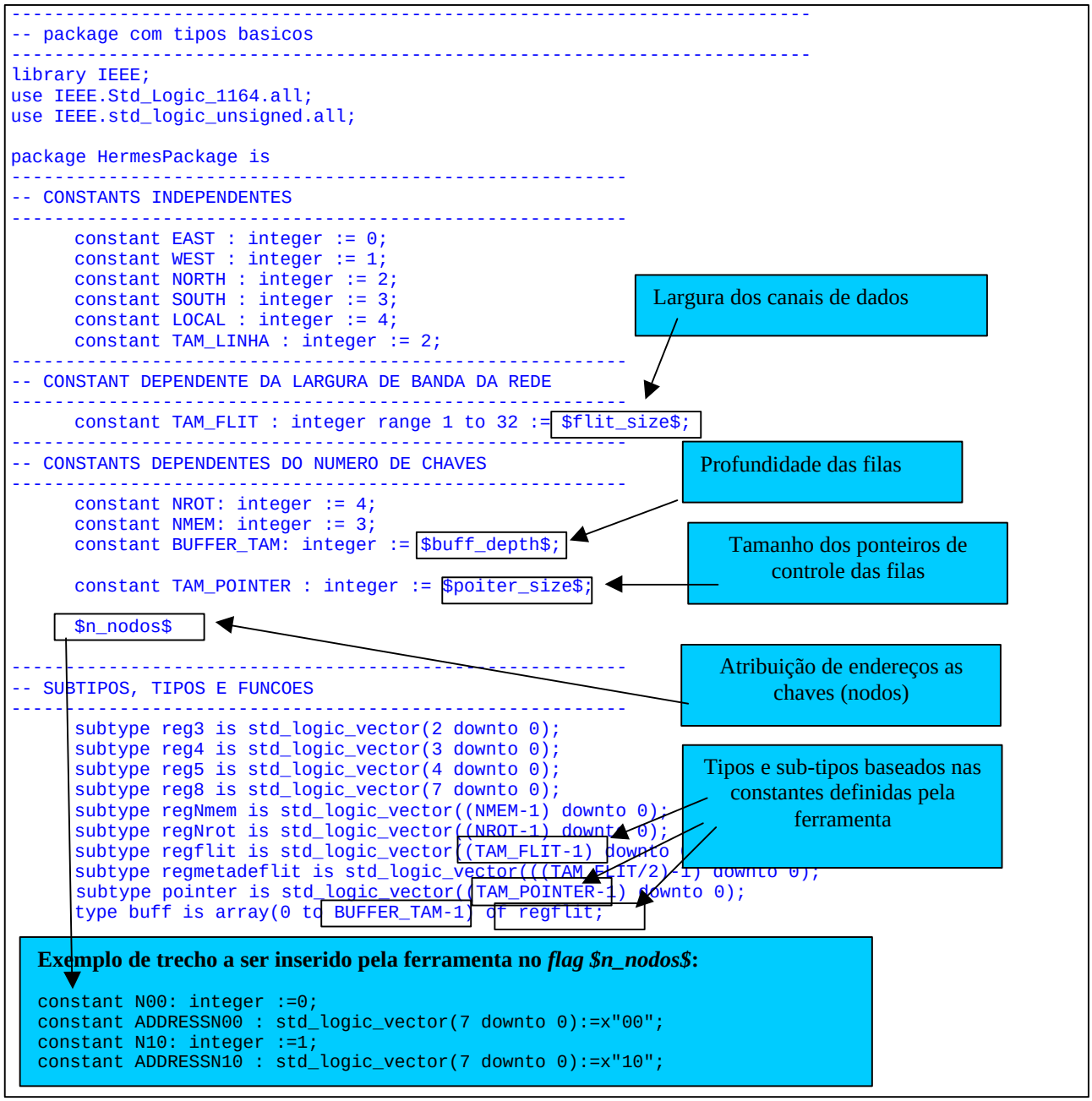


Figura 50 - Trecho do código VHDL da biblioteca *Hermes_Package*.

Neste arquivo foram inseridos *flags* que indicam pontos onde a ferramenta deve inserir os parâmetros escolhidos pelo usuário. O arquivo *Hermes_Package.vhd* é lido pela ferramenta, que faz a substituição dos *flags* e grava o novo arquivo no diretório de projeto definido pelo usuário. Segue a descrição das *flags* inseridos no código *Hermes_Package.vhd*:

- *flag $flit_size$* - pode ser substituído pelos valores 8, 16, 32 ou 64 (*bits*), correspondente à largura do canal de dados.

- *flag $buff_depht$* - pode receber os valores 4, 8, 16 ou 32, de acordo com a profundidade da fila definida pelo usuário.

- *flag $poiter_size$* - corresponde ao tamanho dos ponteiros que controlam a utilização das filas. Seu valor é calculado pela ferramenta de acordo com a profundidade das filas, sendo igual a log_2 *$buff_depht$*.

- *flag n_nodos* - é substituído por uma seqüência de endereços das chaves (como mostrado na Figura 50).

Na Figura 50 também é possível observar a existência de tipos e sub-tipos cujos tamanhos são dependentes das constantes definidas nas *flags*. Os demais módulos da rede intra-chip devem ter os seus sinais e portas declaradas com tipos e tamanhos baseados nas constantes geradas na biblioteca Hermes_Package.vhd.

5.1.2 Parametrização da Chave e da Fila

Para parametrização da chave e da fila foram inseridos *flags* nos arquivos *Chave.vhd* (trecho de código ilustrado na Figura 52) e *Fila.vhd* (trecho de código ilustrado na Figura 53).

As *flags* inseridas no arquivo *Chave.vhd* são: (*i*) *$Chave$*, que deve ser substituído pelo nome que define o tipo de chave; (*ii*) *$filas$*, onde serão inseridas (mapeadas) as filas utilizadas de acordo com o tipo de chave; e (*iii*) *$zeros$*, onde são "aterrados" (recebem o valor '0') os sinais que deveriam ser conectados às filas removidas. A Figura 51 mostra os pontos onde ocorre a parametrização nas chaves e nas filas, ou seja, a profundidade (capacidade) das filas e a largura dos canais de dados (que também afeta a largura das filas.

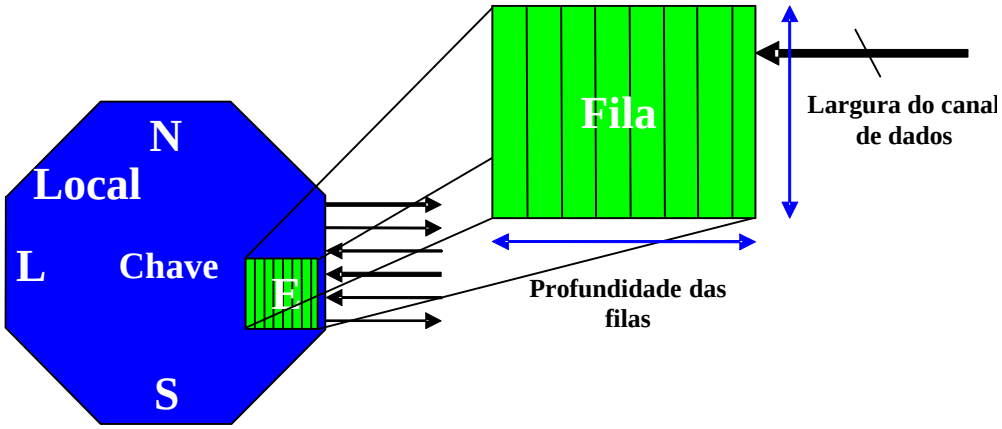


Figura 51 - Pontos parametrizáveis na chave e na filas.

No trecho de código mostrado na Figura 52, para a *flag $zeros$*, pode-se observar que os sinais que seriam conectados à fila leste, índice '1', são conectados à zero ("aterrados"). Já no trecho de substituição ao *flag $filas$*, observa-se o mapeamento da fila oeste, índice '0', utilizada por esta chave. As outras filas utilizadas seriam mapeadas logo a seguir. A *Hermes_Chave.vhd* serve como base para a geração dos diferentes tipos de chaves. Os tipos de chaves, bem como a utilização e remoção de filas são discutidas na Seção 5.1.3.

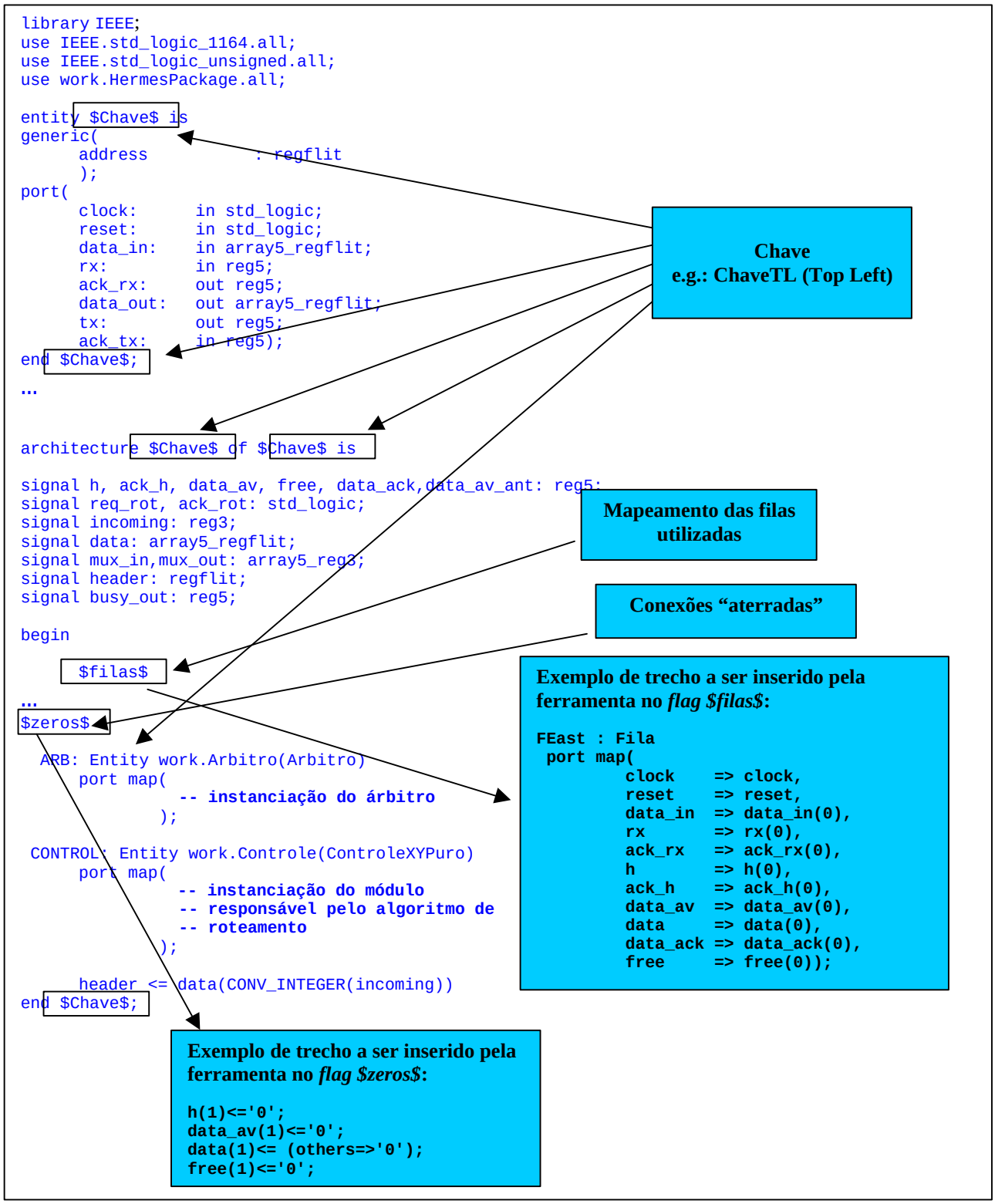


Figura 52 - Trecho do código do arquivo *Hermes_Chave.vhd*.

O terceiro arquivo parametrizado foi o *Fila.vhd*. Neste arquivo não foram inseridos *flags*, porém, alguns trechos precisaram ser modificados. A Figura 53 apresenta um trecho deste código. Neste trecho pode-se observar as declarações de sinais e portas com tamanhos baseados nas constantes geradas na biblioteca *Hermes_Package.vhd*, bem como comparações utilizando estas constantes. Antes das alterações, estes sinais e comparações utilizavam valores numéricos em vez de constantes.

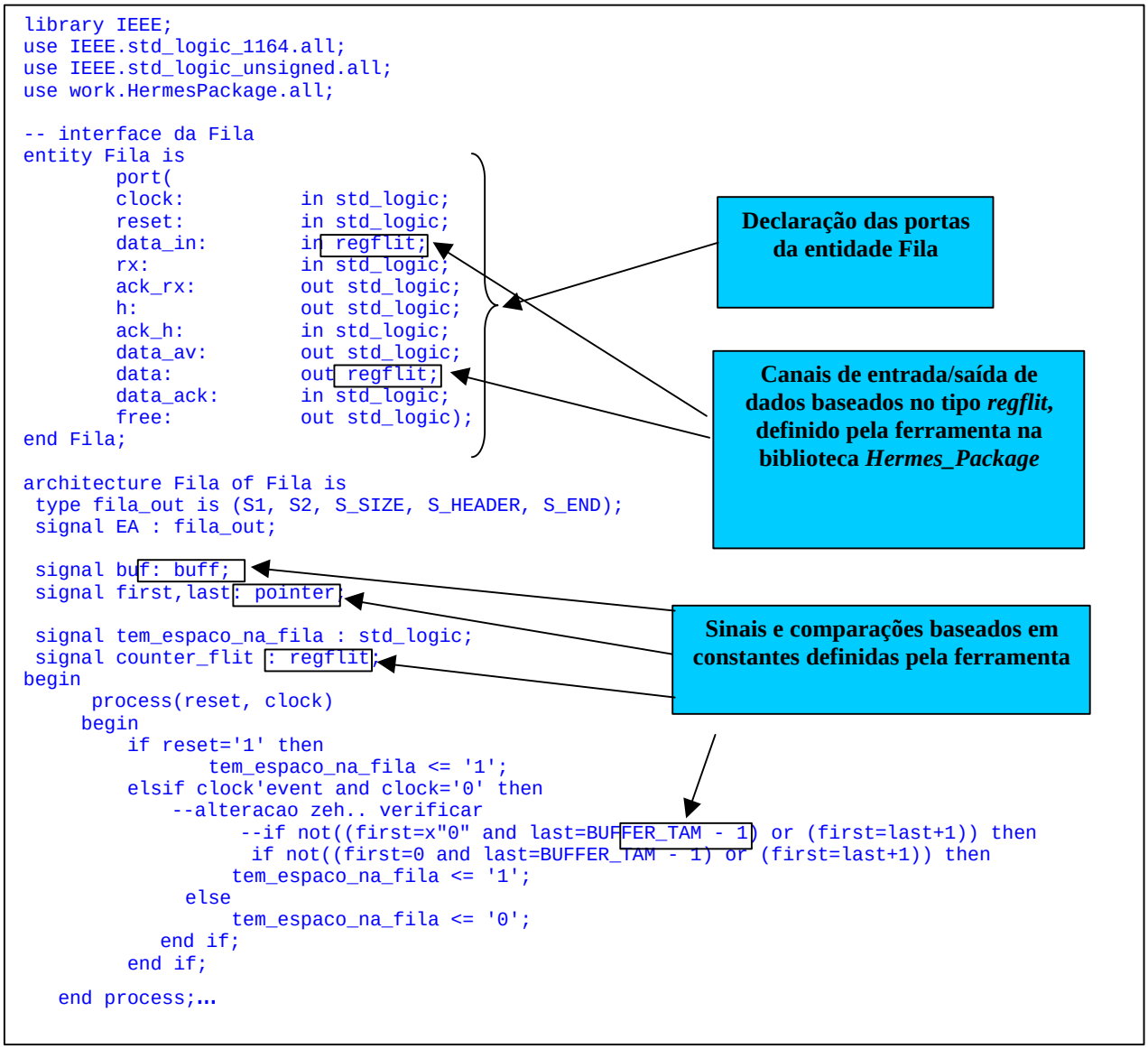


Figura 53 - Trecho do código parametrizável do arquivo *Fila.vhd*.

5.1.3 Topologia da rede

A partir da configuração da Chave, Seção anterior, é possível escolher a topologia mais adequada para a aplicação desejada. Até o presente momento, a ferramenta Maia permite a geração de quatro topologias: malha, torus, torus dobrado, e anel. As topologias torus utilizam todas as portas de conexão das chaves, o que não acontece nas topologias malha e anel.

A Figura 54 ilustra um exemplo de rede intra-chip organizada sob a topologia malha. Nesta topologia as filas não utilizadas devem ser removidas da descrição VHDL a fim de reduzir a área da rede intra-chip. É o caso da chave presente na última coluna da última linha, que possui dois canais sem conexão (sul e oeste). O tipo de chave é definido pela sua posição na rede. Por exemplo, a chave mostrada na Figura 54 é classificada como ChaveBR (*Bottom Right*), ou seja, pertence à última linha e à coluna a direita da rede intra-chip. Desta forma, são nove os tipos de chaves: *Top Left* (TL), *Top Center* (TC), Top Right (TR), *Center Left* (CL), *Center* (CC), *Center Right* (CR), *Bottom Left* (BL), *Bottom Center* (BC) e *Bottom Right* (BR).

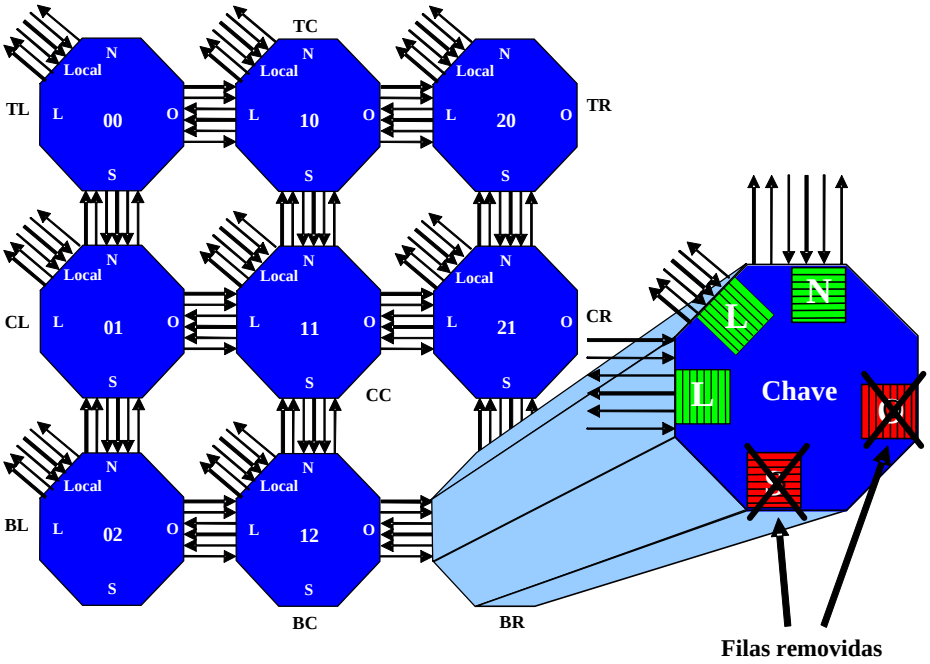


Figura 54 - Descrição dos nove tipos de chaves em uma topologia malha e um exemplo de chave onde remoção de filas deve ocorrer.

A remoção de filas não utilizadas veio a colaborar para a redução de área de chaves que não possuem conexões em todas as portas. A Tabela 10 mostra o consumo de área de uma HERMES 3 x 3 (com 9 chaves), com largura do canal de dados de 8 bits, filas com profundidade de 8 palavras, prototipada na plataforma com o FPGA XC2V1000. Foram feitas duas implementações dessa HERMES: (*i*) todas as cinco filas com todas chaves, com um consumo de 57% do dispositivo; (*ii*) HERMES 3 x 3 gerada pela ferramenta Maia e, portanto, sem as filas da periferia (filas removidas), obtendo-se um consumo de 42% do dispositivo. Com base nestes dados, fica claro que a redução de área obtida com a remoção das filas sem utilização é importante, dado que a fila é o componente que mais consome área na chave.

Tabela 10 - Consumo de área de duas implementação de uma HERMES 3 x 3: uma com todas a filas, e outra apenas com a filas utilizadas.

Implementação	Custo (área)
Normal, com aterramento das filas não utilizadas.	2932 dos 5120 *Slices* do dispositivo XC2V1000 (57%)
NoC gerada pela ferramenta, com remoção de filas não utilizadas.	2191 dos 5120 *Slices* do dispositivo XC2V1000 (42%)

A Figura 55 ilustra o arquivo de mais alto nível na hierarquia da rede intra-chip, denominado de *NoC.vhd*. Neste trecho de código pode-se observar os mapeamentos dos diferentes tipos de chaves e as conexões dos sinais, fazendo com que as saídas de uma chave conecte-se às entradas de outra chave adjacente.

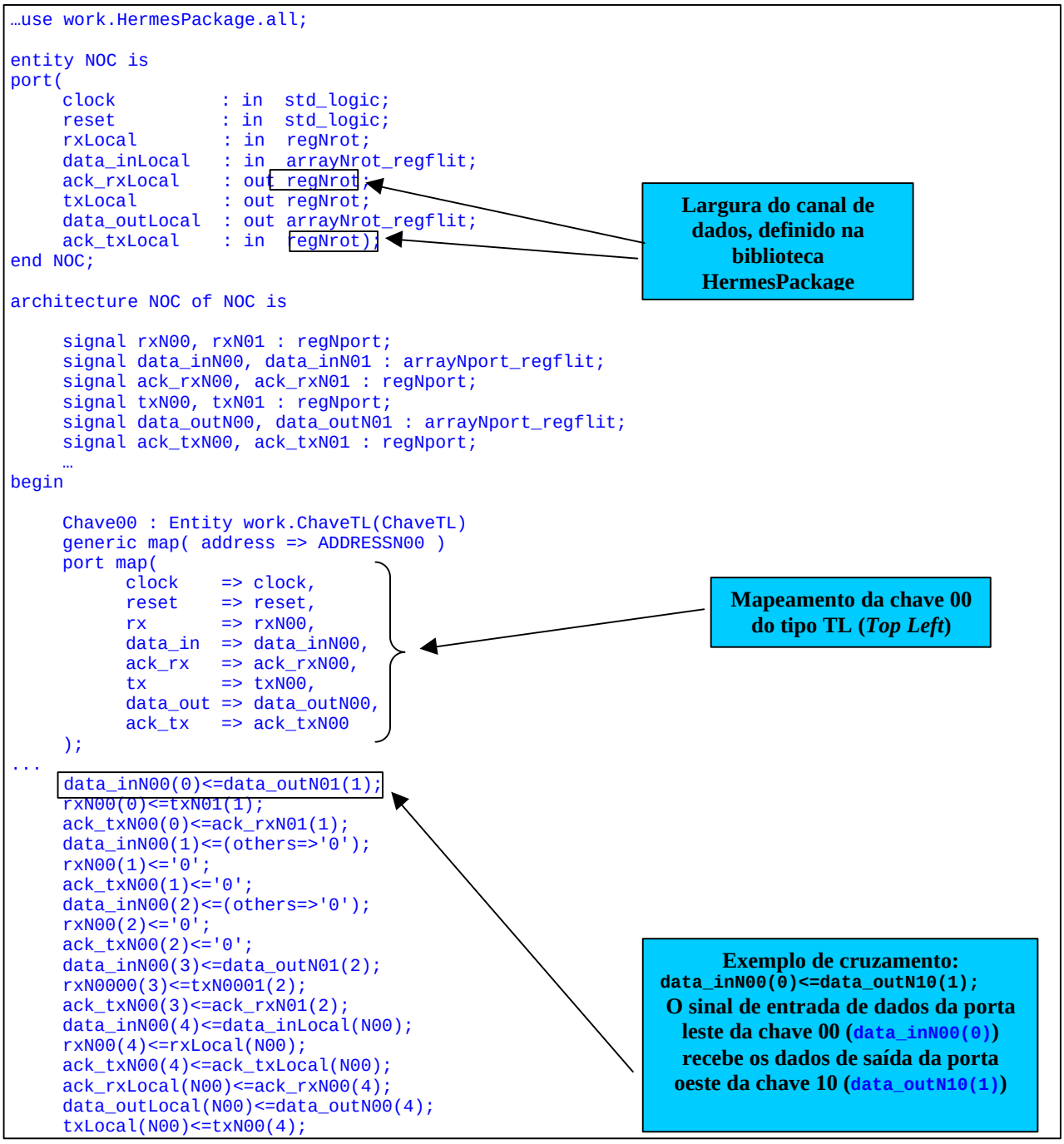


Figura 55 - Trecho do código VHDL da entidade NoC que descreve as portas de entrada/saída e os sinais que conectam as chaves.

5.2 Interface e funcionalidades da Ferramenta Maia

A interface da ferramenta Maia é dividida em quatro blocos básicos: (*i*) área de visualização; (*ii*) barra de parametrização da rede; (*iii*) menu; e (*iv*) *log* de processos. Esta interface é apresentada na Figura 56.

A *área de visualização* representa a rede de acordo com os valores contidos nos campos de parametrização da rede (índice 2 na Figura 56). Com isso, o usuário pode visualizar a dimensão da rede, os endereços, as conexões de cada chaves. Além disso, é possível escolher o tipo de IR de cada chave, clicando sobre a área da mesma (caso a opção *Interface OCP* esteja selecionada). Na

Figura 56 a área de visualização representada uma rede malha de dimensão 3x3, sendo essa composta por três chaves com IRs-E (00, 20, 12), quatro chaves com IRs-M (10, 11, 02, 22) e duas chaves com IRs-ME (01, 21).

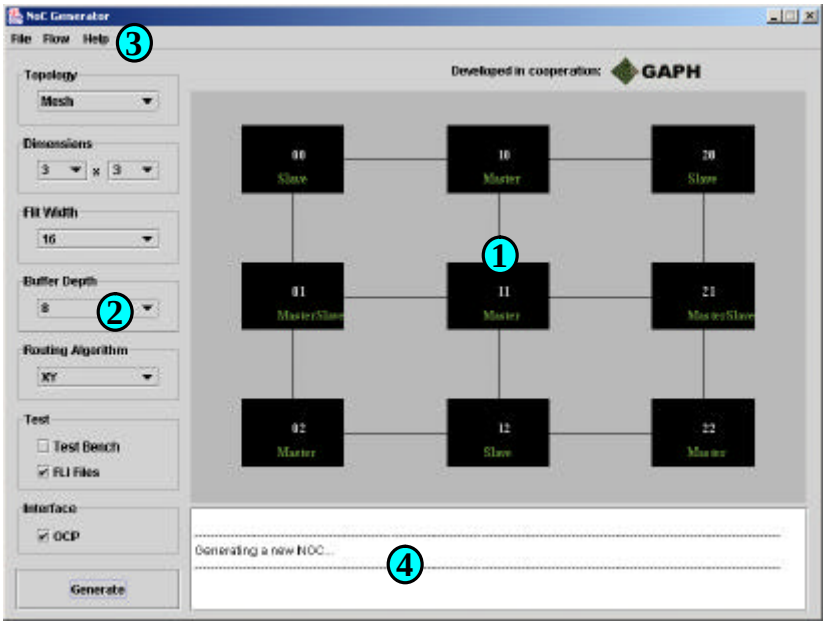


Figura 56 - Interface da ferramenta Maia.

O usuário parametriza a rede a partir do campos encontrados na *barra de parametrização* (índice 2 na Figura 56). A partir desses campos é possível configurar: (*i*) a largura do *flit*; (*ii*) profundidade das filas de entrada das chaves; (*iii*) a dimensão da rede; (*iv*) a topologia. No caso da topologia malha o usuário pode escolher o algoritmo de roteamento para sua rede. Com isso, torna-se possível verificar o desempenho do mesmo frente a sua aplicação. Dentre os algoritmos suportados pela ferramenta citam-se: (*i*) XY puro; (*ii*) *West-First Minimal*; (*iii*) *West-First Non-Minimal*; e (*iv*) *Negative-First Non-Minimal*. As particularidades inerentes a esses algoritmos de roteamento são descritos em [GLA94]. A implementação de algoritmos livre de *deadlock* para as topologias torus e torus dobrado está em andamento. Até o presente momento a ferramenta Maia apenas permite a geração de redes intra-chip baseadas na estrutura HERMES. O suporte para geração de redes baseadas na estrutura da rede SoCIN [ZEF03] também está em andamento.

Ainda na *barra de configuração* pode-se selecionar o tipo de *testbench* a ser gerado. A ferramenta Maia suporta a geração de dois tipos de *testbench*: descrito em VHDL ou implementado com *FLI*. *FLI* é uma interface de comunicação que permite descrever parte de um sistema em linguagem C e a outra parte em linguagem VHDL. No presente caso, a rede-IP é descrita em VHDL e o *testbench* em C. O simulador que suporta de forma nativa esta co-simulação *ModelSim* [MOD02]. Para facilitar o processo de simulação é gerado um *script* que pode ser utilizado para compilação e simulação do sistema no simulador *ModelSim*. Este *script* informa ao simulador *ModelSim* a ordem de compilação dos arquivos e o tempo de simulação do sistema (rede-IP e *testbench* gerados). O *testbench*, nos dois formatos, consiste de leituras e escritas de/em arquivos, onde cada arquivo de entrada possui os pacotes que um núcleo deve enviar para a rede. Os arquivos de entrada recebem o prefixo *in* mais o endereço correspondente à posição do núcleo na rede (e.g. in00.txt, arquivo de entrada do núcleo na posição X= 0 e Y=0). Por sua vez, os arquivos de saída apresentam os pacotes que chegaram a um determinado núcleo ao qual o arquivo está associado. Estes arquivos recebem o prefixo *out* mais o endereço da posição do núcleo (e.g. out00.txt). Caso

deseje-se utilizar outro simulador, como por exemplo o *ActiveVHDL*, é responsabilidade do usuário criar e hierarquizar o projeto.

A barra de menu (índice 3 na Figura 56) possui três menus: *File*, *Flow* e *Help*. O menu *File* possui a opção de criar um novo projeto, onde se determina o nome e o diretório de trabalho do mesmo, ou seja onde será gerada a rede-IP e o *testbench* (se for o caso). O menu *Flow* é composto por três itens: (*i*) *Traffic Generation*; (*ii*) *Simulation*; e (*iii*) *Traffic Analyser*.

O módulo *Traffic Generation* gera três formatos de arquivos de entrada, um para o *testbech* descrito em VHDL, um segundo para o *testbench* FLI e um terceiro no formato STL, utilizados na ferramenta *CoreCreator*. A partir da interface desse módulo, conforme ilustrado na Figura 57, é permitido escolher a dimensão da rede (parâmetro herdado da janela principal), a largura do *flit* (8, 16, 32 e 64 bits, também herdado da janela principal), o número de pacotes a serem enviados por cada núcleo, o número de *flits* por pacote, o destino dos pacotes e número de vezes que será gerado o tráfego. O índice 1 da Figura 57 informa a geração de tráfego para 9 núcleos (dimensão 3x3), enquanto que o índice 2 informa a geração de 20 pacotes por núcleo, sendo que cada pacote será composto por 100 *flits* e o destino destes pacotes serão determinados aleatoriamente. Por fim, o índice 3 da Figura 57 informa a geração de tráfego apenas para o formato *FLI*.

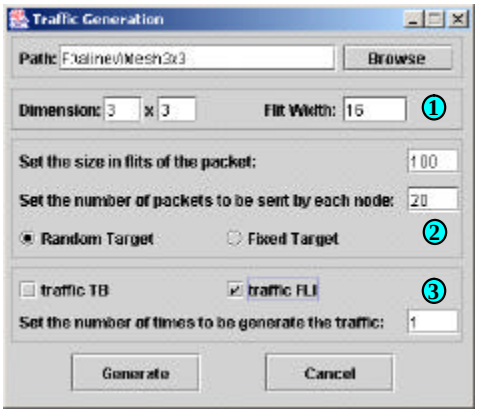


Figura 57 - Interface do módulo *Traffic Generation*.

O arquivo de tráfego é um conjunto de valores hexadecimais gerados aleatoriamente pela ferramenta. Estes valores hexadecimais são agrupados em pacotes que seguem o protocolo de comunicação especificado na Seção 4.2. Após gerar o tráfego é possível testar o sistema através do item do menu *Simulation*. Ao clicar este item executam-se os *scripts* do *testbench* (no caso da ferramenta *Modelsim*) e ao final desse processo são gerados os arquivos de saídas. Devido à complexidade de visualizar os *flits* de um pacote da chave origem até o seu destino, optou-se pelo desenvolvimento de um módulo que permitisse analisar o tráfego da rede. Este módulo é denominado de *Traffic Analyser*.

A partir dos arquivos de saída gerados pelo processo de simulação, o módulo *Traffic Analyser* verifica se todos os pacotes enviados foram recebidos em seus respectivos destinos. A Figura 58 ilustra a interface do módulo *Traffic Analyser*.

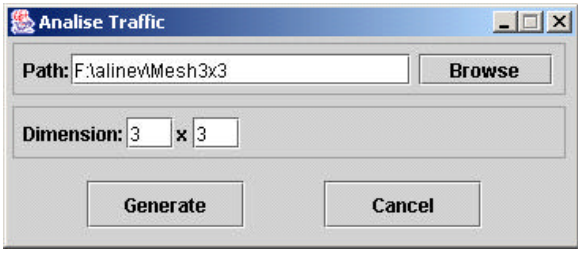


Figura 58 - Interface do módulo *Traffic Analyser*.

Além disso, este módulo gera um arquivo, denominado *analise.txt*, que apresenta as seguintes estatísticas: (*i*) número de pacotes recebidos por cada nodo da rede; (*ii*) número total de pacotes recebidos; (*iii*) tempo mínimo, médio e máximo de transmissão de um pacote; (*iv*) desvio padrão do tempo de transmissão de um pacote e (*v*) tempo total de transmissão de todos os pacotes. A partir desses resultados é possível comparar diferentes configurações de redes intra-chip para um determinado fluxo de dados, como ilustrado na Figura 59. A Figura 59 apresenta o exemplo de relatório gerado a partir da análise de uma rede intra-chip com topologia malha de dimensão 5x5.

```
Diretório: C:\Ost\eu\Mesh5x5_16F8B_WFM
Dimensão: 5x5

número de pacotes recebidos pelo núcleo 0 = 15
número de pacotes recebidos pelo núcleo 1 = 16
número de pacotes recebidos pelo núcleo 2 = 19
número de pacotes recebidos pelo núcleo 3 = 21
número de pacotes recebidos pelo núcleo 4 = 27
número de pacotes recebidos pelo núcleo 5 = 23
número de pacotes recebidos pelo núcleo 6 = 23
número de pacotes recebidos pelo núcleo 7 = 17
número de pacotes recebidos pelo núcleo 8 = 10
número de pacotes recebidos pelo núcleo 9 = 27
número de pacotes recebidos pelo núcleo 10 = 20
número de pacotes recebidos pelo núcleo 11 = 19
número de pacotes recebidos pelo núcleo 12 = 17
número de pacotes recebidos pelo núcleo 13 = 16
número de pacotes recebidos pelo núcleo 14 = 22
número de pacotes recebidos pelo núcleo 15 = 17
número de pacotes recebidos pelo núcleo 16 = 15
número de pacotes recebidos pelo núcleo 17 = 27
número de pacotes recebidos pelo núcleo 18 = 18
número de pacotes recebidos pelo núcleo 19 = 14
número de pacotes recebidos pelo núcleo 20 = 23
número de pacotes recebidos pelo núcleo 21 = 31
número de pacotes recebidos pelo núcleo 22 = 21
número de pacotes recebidos pelo núcleo 23 = 29
número de pacotes recebidos pelo núcleo 24 = 13

total de pacotes recebidos=500
média = 225.77ns # tempo médio para entre de todos os pacotes
mínimo = 13ns    # tempo mínimo para a recepção de um pacote
máximo = 2096ns  # tempo mínimo para a recepção de um pacote
desvio padrão = 260.0362ns
tempo total = 3471ns
```

Figura 59 - Exemplo de um relatório gerado após a análise de tráfego.

Outra funcionalidade apresentada pela ferramenta é a geração dos arquivos necessários para validar uma rede com IRs-OCP. Os arquivos gerados são: (*i*) descrição da interface OCP; (*ii*) arquivo que contém os nomes dos módulos a serem simulados; e (*iii*) arquivos de tráfego baseados na sintaxe da linguagem STL (já descritos anteriormente).

O último bloco que compõe a interface da ferramenta Maia é o *log de processos* (índice 4 na Figura 56) que serve para informar o usuário das operações que estão sendo executadas pela ferramenta.

5.3 Considerações Finais do Capítulo

O produto final da ferramenta Maia consiste em *redes-IP* geradas automaticamente em funções das restrições da aplicação alvo, com interface de rede padronizada (se desejado pelo usuário). Com essa abordagem o usuário deve ser preocupar apenas com as conexões entre os seus núcleos e as portas da rede (como ilustrado na Figura 3, página 6). Ou seja, todo o funcionamento e a interconexão entre as chaves podem ser assumidos como válidas.

Deve-se ressaltar, que todas as funcionalidades descritas acima, hoje suportadas pela ferramenta, foram primeiramente validadas e depois inseridas na ferramenta em questão. Até o presente momento, foram prototipadas em FPGA *redes-IP* geradas pela Maia com largura de canal e *flits* de 8 e 16 bits. No Capítulo 6, é descrito um estudo de caso que utiliza uma *rede-IP* gerada pela ferramenta Maia como meio de interconexão dos núcleos.

Este Capítulo descreveu as principais funcionalidades da ferramenta Maia. Este ferramenta automatizou a geração de redes intra-chip baseadas na infra-estrutura HERMES. A geração de redes intra-chip através da ferramenta traz os seguintes benefícios: (*i*) redução do tempo de projeto; (*ii*) unificação das versões dos códigos dos componentes, simplificando o controle de versões; (*iii*) facilidade para geração de diferentes redes intra-chip para a mesma aplicação, permitindo explorar o espaço de soluções para o problema; (*iv*) geração automática de tráfego e de *testbenchs*, produzindo arquivos de teste a rede; (*vi*) verificação do tráfego na rede, ou seja, se todos os pacotes enviados são recebidos no seus respectivos destinos; e (*vii*) geração do template VHDL de nível hierárquico superior, para conexão de núcleos à rede-IP.

6 ESTUDO DE CASO

Visando validar a interconexão e a comunicação de núcleos com interface padrão OCP via HERMES-OCP foram desenvolvidos dois sistemas como estudos de caso: (*i*) sistema composto por quatro processadores MET2[1] com IR-ME e uma rede HERMES-OCP; e (*ii*) sistema com por duas memórias *BlockRAM* com IR-E, um módulo serial com IR-M, um processador com IR-ME e uma rede HERMES-OCP. Este sistema foi denominado de *SR8-OCP*.

Neste Capítulo, apresenta-se detalhadamente o estudo de caso sistema *SR8-OCP*. Justifica-se essa escolha dado a presença dos três tipos de IRs (mestre, escravo e mestre-escravo). Objetiva-se com isso, demonstrar os passos do processo de integração necessários para interconectar núcleos com interface padrão OCP à rede HERMES-OCP. Cabe ainda salientar, que ambos os estudos (*i*) e (*ii*), foram prototipados na plataforma *Memec Insight* que contém um dispositivo FPGA Virtex-II 1000.

6.1 Processo de integração e descrição do sistema

O primeiro passo para o desenvolvimento do estudo de caso *SR8-OCP* foi a definição dos parâmetros, da rede HERMES-OCP. Os parâmetros adotados para gerar essa rede com IRs OCP são: (*i*) largura de *flit* igual a 16 bits; (*ii*) filas com profundidade de 8 posições; (*iii*) algoritmo de roteamento XY puro; (*iv*) topologia malha de dimensão 2x2; e (*v*) duas IRs mestre OCP, uma IR escravo OCP e uma IR mestre-escravo OCP. Definidos os parâmetros da rede, e conseqüentemente, as particularidades inerentes às interfaces OCP (IRs) inicializou-se o desenvolvimento das interfaces de rede OCP de cada um dos núcleos que devem compor o sistema. De posse da rede HERMES-OCP e dos núcleos com interface OCP o projetista (integrador do sistema) deve garantir o funcionamento do sistema como um todo, após a interconexão de todos os módulos do sistema (núcleos e a rede HERMES-OCP).

No presente trabalho, adotaram-se os seguintes passos para interconectar os núcleos à rede intra-chip: (*i*) desenvolvimento da(s) IR(s) OCP do lado dos núcleos (*ii*) validação da(s) IR(s) OCP via *CoreCreator*; (*iii*) validação funcional por simulação da integração do núcleo à rede HERMES-OCP e (*iv*) prototipação. A Figura 60 ilustra a estrutura do sistema *R8-OCP*.

[1] O MET2 é um processador de testes programável que pode ser utilizado para gerar/testar valores de testes pseudo-randômicos. A descrição desse processador ser encontrada em [AMO03].

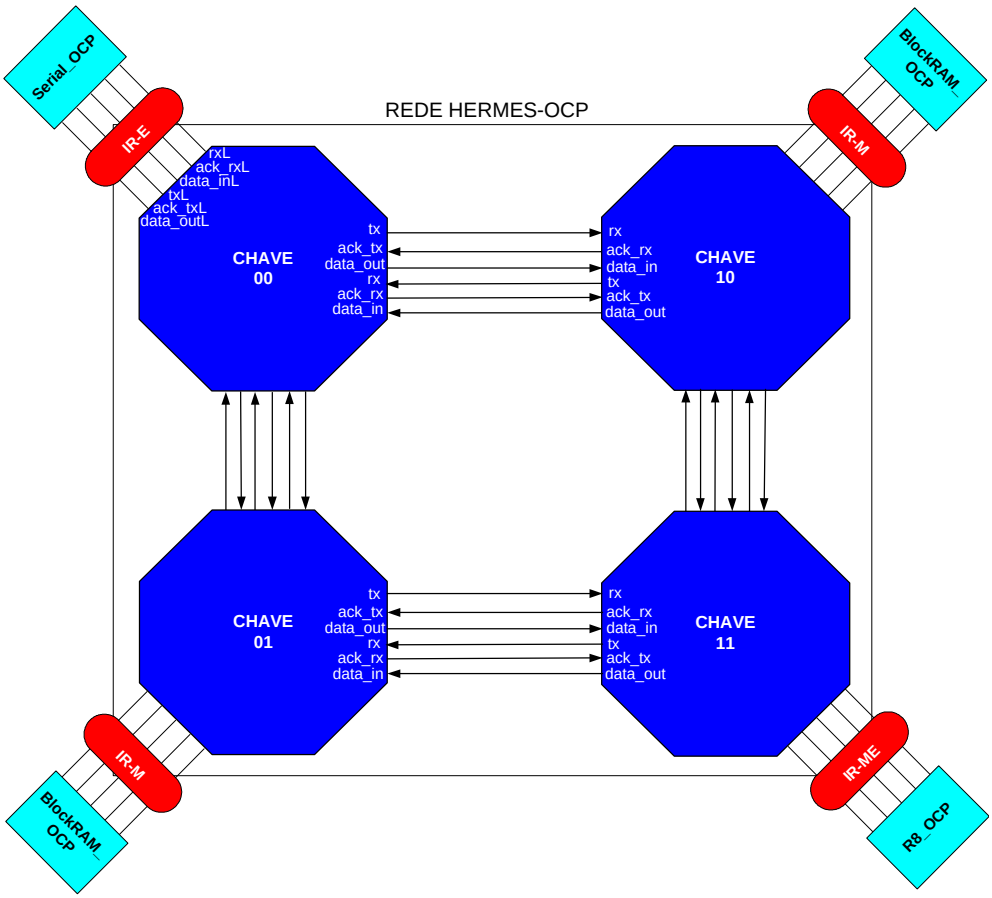


Figura 60 – Estrutura do sistema escolhido como estudo de caso.

6.2 Núcleos do *SR8-OCP*

Todos os núcleos que compõe a estrutura do estudo de caso em questão foram desenvolvidos no grupo de atuação. Para cada um dos núcleos foram desenvolvidos invólucros OCP (no contexto dessa dissertação, IRs). Cabe ainda salientar que esses invólucros foram validados através da utilização da ferramenta *CoreCreator*.

6.2.1 Núcleo BlockRam_OCP

O núcleo básico de memória disponível no FPGA Virtex II é denominado *BlockRAM*, composto por 18 kbits. Cada *BlockRAM* corresponde à altura de 4 CLBs, estando posicionadas nas laterais esquerda e direita do FPGA. A Figura 61 ilustra os sinais desse núcleo, que são descritos abaixo, e os sinais da IR-E OCP (Escravo OCP) desenvolvida.

- *clk: relógio da memória.*
- *rst*: quando ativo inicializa as posições de memória com zeros.
- *en*: quando ativo habilita a *BlockRAM* para leitura ou escrita.
- *we*: quando ativo habilita a escrita na *BlockRAM.*
- *addr*: informa o endereço de leitura ou escrita.

- *di*: informa o valor a ser escrito no endereço *addr*.
- *do*: informa o valor lido do endereço *addr*.

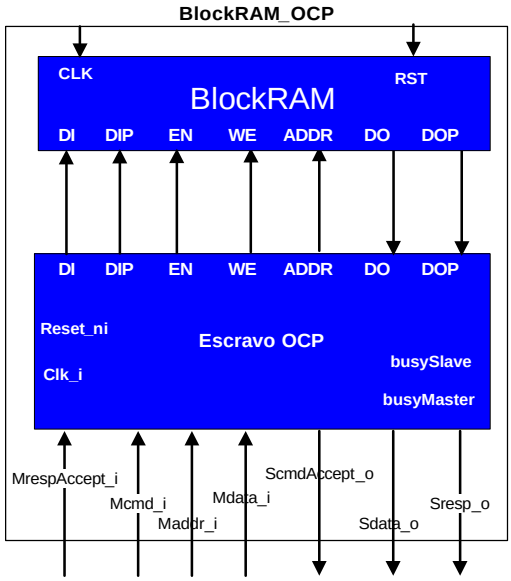
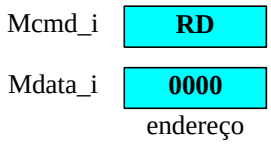


Figura 61 - Componentes do invólucro *BlockRam_OCP*.

Esta memória pode receber dados (pela IR-E OCP) provindo de um núcleo mestre interconectado à HERMES-OCP em quatro formatos diferentes. Os comandos suportados até o presente momento são: (*i*) *read simples*; (*ii*) *write simples*; (*iii*) *write burst modo* 1; (*iv*) *write burst modo* 2.

6.2.1.1 Read Simples

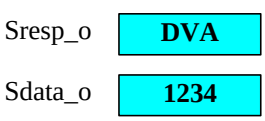
A memória recebe o comando *read simples* do núcleo que atua como mestre, no seguinte formato:



Onde: *Mcmd_i* informa o comando OCP da mensagem e *Mdata_i* informa o endereço da memória a ser lido.

Figura 62 - Formato da mensagem recebida pela *BlockRAM_OCP* do mestre.

Por sua vez, a memória responde ao núcleo mestre transmitindo dados solicitados no seguinte formato:



Onde, *Sresp_o* informa que o dado lido pelo comando RD está disponível e o sinal *Sdata_o* contém o dado lido.

Figura 63 - Formato da mensagem, contendo os dados solicitados, transmitida para o mestre.

6.2.1.2 Write Simples

A memória recebe o comando *write simples* do núcleo que atua como mestre, no seguinte

formato:

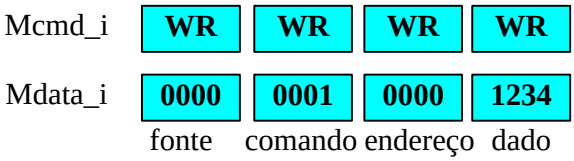


Figura 64 - Formato da mensagem recebida pela memória.

Onde: *Mcmd_i* informa o comando OCP da mensagem. O sinal *Mdata_i* recebe na ordem: (*i*) endereço do núcleo origem da mensagem (fonte); (*ii*) o comando da mensagem, sendo que o comando write simples é identificado pelo valor 1 (um); (*iii*) o endereço da memória onde o dado será escrito; e (*iv*) o dado a ser escrito.

6.2.1.3 Write Burst modo 1

A memória recebe o comando *write burst modo* 1 de um núcleo que atua como mestre, no seguinte formato:

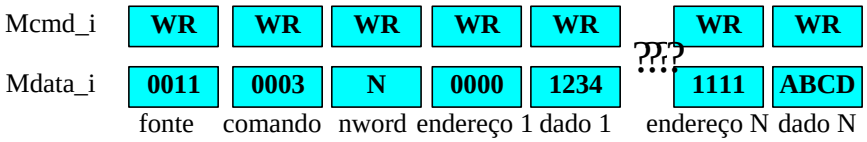


Figura 65 - Formato da mensagem recebida pela memória.

Onde: *Mcmd_i* informa o comando OCP da mensagem. O sinal *Mdata_i* recebe na ordem: (*i*) endereço do núcleo origem da mensagem (fonte); (*ii*) o comando da mensagem, sendo que o comando write burst modo 1 é identificado pelo valor 3 (três). Este comando possibilita que N dados sejam escritos a partir da informação do número de pares endereço-dado; (*iii*) o terceiro *flit* contém o número de dados a serem escritos (nword); (*iv*) o quarto *flit* contém o endereço da memória onde será escrito o primeiro dado; (*v*) *flit* contendo o primeiro dado a ser escrito. Os próximos pares de *flits* correspondem a um endereço e o dado a ser escrito.

6.2.1.4 Write Burst modo 2

O comando *write burst modo* 2 possibilita que N dados sejam escritos a partir da informação de um endereço inicial, onde N é o número de dados a serem escritos. A memória recebe o comando *write burst modo* 2 de um núcleo que atua como mestre, no seguinte formato:

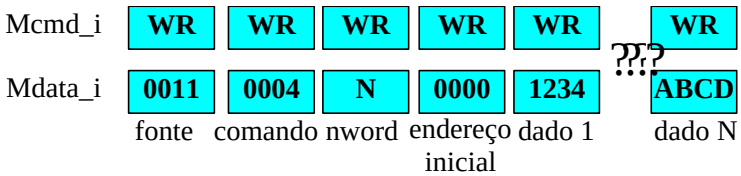


Figura 66 - Formato da mensagem recebida pela memória.

Onde: *Mcmd_i* informa o comando OCP da mensagem. O sinal *Mdata_i* recebe na ordem: (*i*) endereço do núcleo origem da mensagem (fonte); (*ii*) o comando da mensagem, sendo que o comando *write burst modo* 2 é identificado pelo valor 4 (quatro); (*iii*) o terceiro *flit* contém o número de dados a serem escritos (nword); (*iv*) o quarto *flit* contém o endereço inicial da memória

onde serão escritos os dados; (*v*) *flit* contendo o primeiro dado a ser escrito. Os próximos *flits* correspondem a um dado que deve ser escrito.

A Máquina de estados responsável pela implementação dos 4 comandos descritos acima é apresentada no Anexo I, Figura 85.

6.2.2 Núcleo Serial_OCP

Os sinais dispostos na parte superior da Figura 67 fazem interface com o computador hospedeiro, que envia e recebe dados bit a bit. Os sinais dispostos na parte inferior da figura fazem interface com o sistema, que envia e recebe dados byte a byte. Abaixo segue uma breve explicação sobre cada um dos sinais ilustrados na Figura 67.

- *rxd:* envia dados bit a bit pela serial.
- *txd:* recebe dados bit a bit pela serial.
- *rx_data:* byte a ser convertido e transmitido para o computador hospedeiro.
- *rx_start:* indica que existe um byte disponível a ser transmitido em *rx_data*.
- *rx_busy:* indica se está sendo efetuada uma transmissão.
- *tx_data:* byte recebido do computador hospedeiro.
- *tx_av:* indica que existe um byte disponível no *tx_data*.

A Serial foi envolvida por um invólucro que contém uma interface mestre OCP, conforme ilustrado na Figura 67.

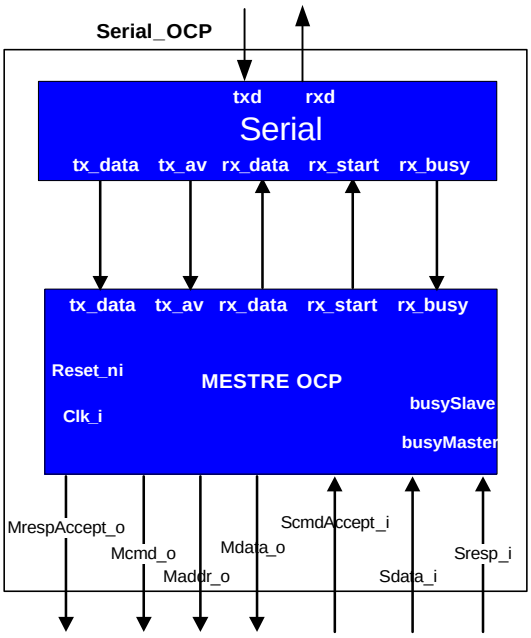


Figura 67 - Componentes do invólucro Serial_OCP.

O núcleo Serial_OCP recebe dados a partir de um programa executando em um

computador hospedeiro (denominado software serial). São cinco os formatos permitidos, dependendo do comando recebido. Estes comandos são: (*i*) *read simples*; (*ii*) *write simples*; (*iii*) *write burst modo* 1, (*iv*) *write burst modo* 2; e (*v*) *reset*.

6.2.2.1 Comando Read Simples

O núcleo Serial_OCP recebe o comando *read simples* do software serial no formato:

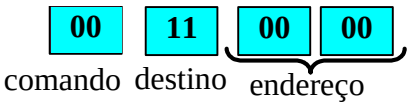


Figura 68 - Formato da mensagem recebida do software serial.

Onde: (*i*) *comando*, sendo *read simples* identificado pelo valor 0 (zero); (*ii*) endereço de destino da mensagem; (*iii*) endereço interno ao núcleo onde será realizada a leitura. O formato da mensagem transmitida da Serial_OCP para um núcleo escravo é ilustrada na Figura 69:

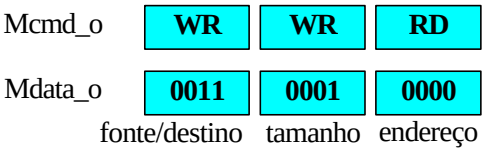


Figura 69 - Formato da mensagem transmitida para um núcleo escravo.

Onde: (*i*) o primeiro *flit* (fonte/destino) informa o endereço do núcleo origem (núcleo Serial_OCP) nos 8 bits mais significativos e o endereço do núcleo destino nos 8 bits menos significativos; (*ii*) o segundo *flit* (tamanho) informa o número de *flits* do payload; (*iii*) e finalmente o último *flit* informa o endereço interno ao núcleo onde será realizada a leitura. Por sua vez, o núcleo Serial_OCP recebe dados do escravo no seguinte formato:

Figura 70 - Formato da mensagem recebida do escravo.

Onde: *Sresp_i* informa que o dado de retorno da leitura está disponível e *Sdata_i* contém o dado referente a leitura (no exemplo, 1234).

Serial_OCP transmite dados para o software serial no formato:

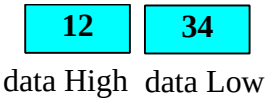


Figura 71 - Formato da mensagem transmitida para o software serial.

Onde: *data High* indica os 8 bits mais significativos do dado lido e *data Low* informa os 8 bits menos significativos do dado lido.

6.2.2.2 Comando Write Simples

O núcleo Serial_OCP recebe o comando *write simples* do software serial no formato:

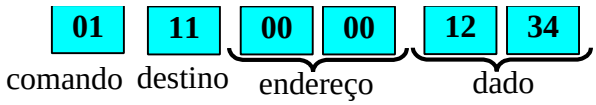


Figura 72 - Formato da mensagem recebida do software serial quando o comando é o *write simples*.

Onde: (*i*) *comando*, refere-se ao *write simples* identificado pelo valor 1 (um); (*ii*) o endereço de *destino* da mensagem; (*iii*) *endereço* interno ao núcleo onde será realizada a escrita; e (*iv*) o *dado* a ser escrito.

Serial_OCP transmite o comando *write simples* para o escravo no formato:

Figura 73 - Formato da mensagem transmitida para o escravo.

Onde: (*i*) o primeiro *flit* (fonte/destino) informa o endereço do núcleo origem (núcleo Serial_OCP) nos 8 bits mais significativos e o endereço do núcleo destino nos 8 bits menos significativos; (*ii*) o segundo *flit* (tamanho) informa o número de *flits* do payload; (*iii*) o terceiro *flit* informa o comando da mensagem; (*iv*) o quarto *flit* informa o endereço interno ao núcleo onde será realizada a escrita; (*v*) o dado a ser escrito.

6.2.2.3 Comando Write Burst modo 1

O núcleo Serial_OCP recebe o comando *write burst modo* 1 do software serial no formato:

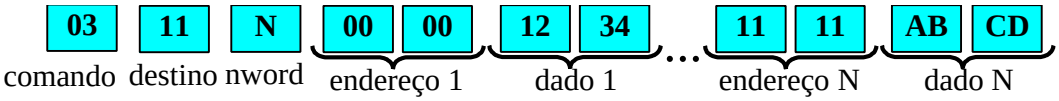


Figura 74 - Formato da mensagem recebida do software serial quando o comando é o *write burst modo* 1.

Onde: (*i*) *comando write burst modo* 1 é identificado pelo valor 3 (três); (*ii*) o segundo *flit* corresponde ao endereço do núcleo destino da mensagem; (*iii*) o terceiro *flit* (nword) informa o número de *flits* a serem escritos; (*iv*) o quarto *flit* informa o *endereço* interno ao núcleo onde será realizada a escrita; e (*v*) o *dado* a ser escrito. O núcleo Serial_OCP transmite o comando *write burst modo* 1 para o escravo no seguinte formato:

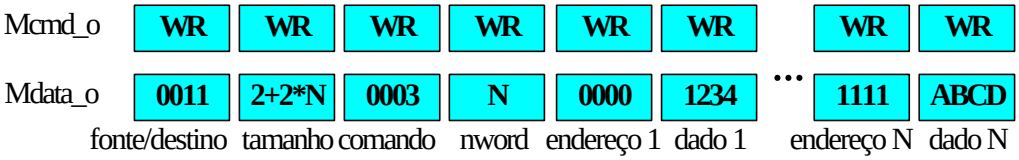


Figura 75 - Formato da mensagem transmitida para o escravo.

Onde: (*i*) o primeiro *flit* (fonte/destino) informa o endereço do núcleo origem (núcleo Serial_OCP) nos 8 bits mais significativos e o endereço do núcleo destino nos 8 bits menos significativos; (*ii*) o segundo *flit* (tamanho) informa o número de *flits* do payload; (*iii*) a terceira transação transmite o endereço interno ao núcleo onde será realizada a leitura; (*iv*) nword informa o número de dados a serem escritos; (*v*) o endereço 1 corresponde ao endereço interno do núcleo onde

será realizada a escrita; e (*vi*) finalmente o *dado* a ser escrito.

6.2.2.4 Comando Write Burst modo 2

O núcleo Serial_OCP recebe o comando *write burst modo* 2 do software serial no seguinte formato:

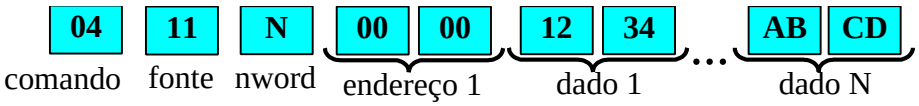


Figura 76 - Formato da mensagem recebida pelo núcleo Serial_OCP do software serial quando o comando é o *write burst modo* 2.

Onde: (*i*) *comando write burst modo* 1 é identificado pelo valor 4 (quatro); (*ii*) o segundo *flit* corresponde ao endereço do núcleo destino da mensagem; (*iii*) o terceiro *flit* (nword) informa o número de *flits* a serem escritos; (*iv*) o quarto *flit* informa o *endereço* interno ao núcleo onde será realizada a escrita; e (*v*) o *dado* a ser escrito. O núcleo Serial-OCP transmite o comando *write burst modo* 2 para o escravo no seguinte formato:

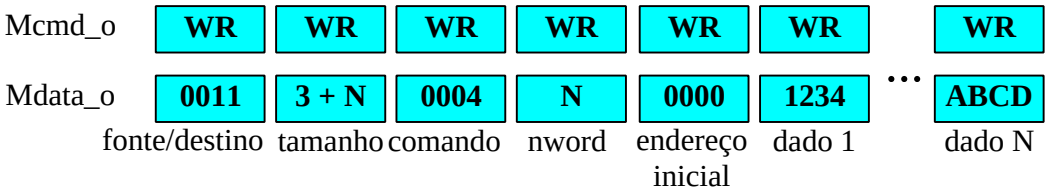


Figura 77 - Formato da mensagem transmitida para o escravo.

6.2.2.5 Comando Reset

O núcleo Serial_OCP recebe o comando *reset* do software serial no seguinte formato:

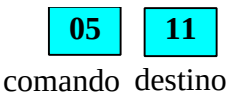


Figura 78 - Formato da mensagem recebida do software serial.

Onde: o comando é identificado pelo valor 5 (cinco) e o destino informa o endereço do núcleo destino a receber a mensagem. O núcleo Serial_OCP transmite o comando reset para o escravo no seguinte formato:

Figura 79 - Formato da mensagem transmitida para o escravo.

Este comando é enviado pelo computador hospedeiro para a inicialização do processador R8.

6.2.3 Núcleo R8_OCP

O núcleo R8_OCP é composto por 4 módulos: (*i*) memória interna (RAM), (*ii*) processador R8, (*iii*) interface Mestre OCP, e (*iv*) interface Escravo OCP. O principal módulo desse núcleo é o processador R8 que é uma organização Von Neumann (memória de dados/instruções unificada), load/store, com CPI entre 3 e 4, barramento de dados e endereços de 16 bits. Esta arquitetura é praticamente uma máquina RISC, faltando, contudo, algumas características gerais de máquina RISC, tal como pipeline e módulos de entrada/saída, como tratamento de interrupções.

O projeto da arquitetura, desenvolvido em VHDL, compreende a descrição de dois blocos principais: o bloco de controle e o bloco de dados. O bloco de controle tem por função gerar os comandos para a busca das instruções e envio de comandos ao bloco de dados para que a instrução seja executada. O bloco de dados contém 16 registradores de uso geral; registradores para armazenamento da instrução corrente (*IR*), endereço da próxima instrução a executar (*PC*) e ponteiro de pilha (*SP*); uma ULA (Unidade Lógica e Aritmética) com 13 operações e 4 qualificadores de estado.

A funcionalidade dos sinais do processador R8 é descrita abaixo:

- *ck*: sincroniza os sinais internos ao processador.
- *rst*: inicializa o processador para iniciar a execução de instruções a partir do endereço zero da memória.
- *ce*: sinal que quando ativo habilita operações na memória.
- *rw*: se for igual a “1” habilita a operação de leitura na memória, “0” habilita operação de escrita.
- *address*: sinal que endereça a memória.
- *dataIn*: sinal de entrada de dados/instruções (16 bits) oriundo da memória ou do barramento.
- *dataOut*: sinal de saída de dados/instruções (16 bits) destinado à memória ou ao barramento.
- *haltR8*: sinal que quando ativo indica o fim do processamento.
- *waitR8*: sinal que quando ativo provoca uma pausa na execução do processador

A interface Mestre OCP é conectada ao processador enquanto a interface Escravo OCP é conectada à RAM interna. A Figura 80 ilustra os quatro componentes que compõe o invólucro R8_OCP. O núcleo R8_OCP é inicializado a partir da recepção do comando *reset* que é recebido através da do módulo Escravo OCP (índice 1 da Figura 80) que ativa o sinal *ResetR8*. Além dos 4 módulos que compõe o núcleo R8 OCP destacam-se um módulo constituído de lógica de controle e três multiplexadores: (*i*) mult2; (*ii*) mult3; (*iii*) mult4; sendo estes representados, respectivamente, pelos índices 2, 3 e 4 da Figura 80. O mult2 é utilizado para verificar se o dado a ser escrito na RAM é o provido pela R8 (dado local) ou o recebido pelo módulo Escravo OCP (dado remoto). O mult3 possui uma funcionalidade semelhante ao mult2, sendo que ao invés do dado seleciona-se o endereço de escrita ou de leitura. Por sua vez, o mult4 verifica se os dados de leitura que estão

chegando, provêm da memória local (RAM) ou de algum núcleo do tipo escravo conectado a rede (através do módulo Mestre OCP). O item 5 e 6 da Figura 80, são utilizados para delegar acesso de escrita ou leitura local (R8) ou remota através do módulo Escravo OCP (outro núcleo do tipo mestre conectado a rede).

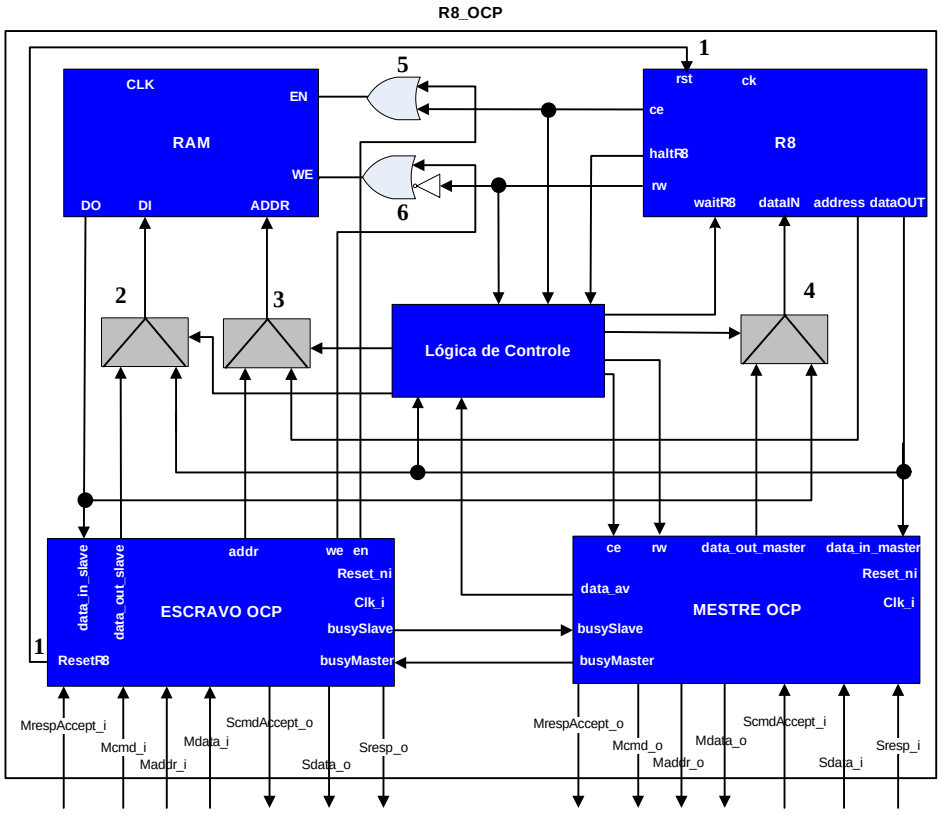


Figura 80 – Componentes do invólucro R8_OCP.

6.3 Validação funcional do Sistema

Esta Seção apresenta a validação dos núcleos Serial_OCP, *BlockRAM*_OCP e o R8_OCP com a HERMES-OCP descrita na Seção anterior.

A primeira etapa do processo de execução corresponde ao envio de um programa de teste do processador R8, através da utilização do programa serial, conforme ilustrado na Figura 81. O núcleo Serial OCP recebe os dados do programa serial, enviando-os ao núcleo R8 OCP. Este programa é recebido pelo módulo Escravo OCP do núcleo R8 OCP e escrito a partir do endereço 0x0000 na memória local (RAM) do processador R8.

O programa de teste é ilustrado na Figura 82. O programa de teste é composto de dois laços. O primeiro laço é responsável por ler o próprio programa de teste da memória local (RAM), enviando-o para o núcleo *BlockRAM*_OCP (chave 10), via o módulo Mestre OCP (linhas 18 a 24 do código acima). Após a escrita na memória remota, gera-se um pacote que contém solicitações de leitura para a *BlockRAM*_OCP, linhas 30-32. Após estas linhas, inicia-se o segundo laço, que corresponde à leitura dos valores da *BlockRAM*_OCP a partir do endereço 0x0000 até o 0x0014, gravando-se os mesmos na memória local (RAM) à partir do endereço 0x0050h, linhas 34-40. O programa grava os dados usando o comando *write burst* (04) e lê usando o comando *read simples*.

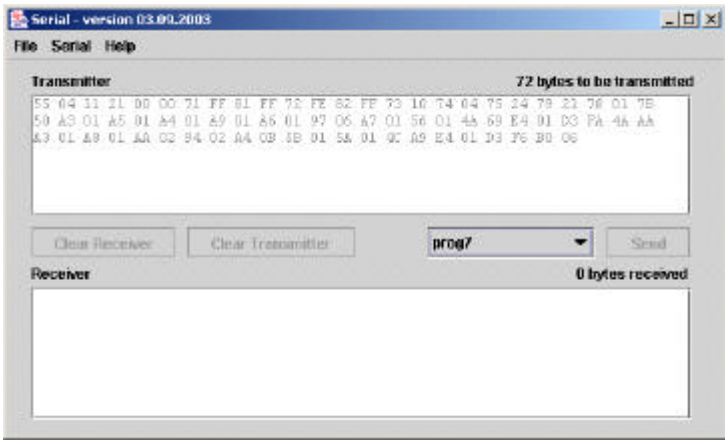


Figura 81 – Tela do programa serial contendo o programa descrito na Figura 82.

```
1.  .CODE
2.  LDL R1,#FFh
3.  LDH R1,#FFh     ;R1 = PORTA DE SAÍDA (0xFFFF)
4.  LDL R2,#FEh
5.  LDH R2,#FFh     ;R2 = PORTA DE ENTRADA (0xFFFE)
6.  LDL R3,#10h     ;DESTINO DO PACOTE (BLOCKRAM CONECTADO AO NÚCLEO 10)
7.  LDL R4,#04      ;COMANDO WRITEBURST2 DA BLOCKRAM
8.  ldl R9,#21h     ;NUMERO DE PALAVRAS A SEREM ESCRITAS
9.  LDL R5,#24h     ;TAMANHO DO PAYLOAD (COMANDO = 1,SIZE = 1,END = 1,DADOS = 21h, TOTAL = 24h)
10. LDL R8,#01h     ;PAYLOAD DO PACOTE DE LEITURA(ENDEREÇO=1)
11. LDL R11,#50h

12. ;-----------------MONTAGEM DO PACOTE-------------------------------------
13. ST R3,R0,R1     ;ENVIA PARA A REDE O DESTINO
14. ST R5,R0,R1     ;ENVIA PARA A REDE O TAMANHO DO PAYLOAD
15. ST R4,R0,R1     ;ENVIA PARA A REDE O COMANDO(Write Burst2)
16. ST R9,R0,R1     ;ENVIA PARA A REDE O NUMERO DE PALAVRAS A SEREM ESCRITAS
17. ST R6,R0,R1     ;ENVIA PARA A REDE O ENDEREÇO INICIAL
18. VOLTA:
19. LD R7,R0,R6     ;BUSCA O DADO NA MEMÓRIA LOCAL
20. ST R7,R0,R1     ;ENVIA PARA A REDE O DADO
21. ADDI R6,#01                                    } o dado corresponde ao próprio programa
22. XOR R10,R6,R9                                  } para a BLOCKRAM.
23. JMPZD #LEITURA
24. JMPD #VOLTA
25. ;---------------------------------------------------------------------

26. LEITURA:
27. XOR R10,R10,R10

28. VOLTA2:

29. ;---------------MONTAGEM DO PACOTE DE LEITURA---------------------------------
30. ST R3,R0,R1     ;ENVIA PARA A REDE O DESTINO
31. ST R8,R0,R1     ;ENVIA PARA A REDE O TAMANHO DO PAYLOAD
32. ST R10,R0,R2    ;ENVIA PARA A REDE O ENDEREÇO E INDICA QUE É UMA LEITURA, POIS USA A PORTA
    DE ENTRADA(0xFFFE)
33. ;---------------------------------------------------------------------

34. LD R4,R0,R2     ;LÊ O DADO DA REDE
35. ST R4,R0,R11    ;GRAVA O DADO LIDO NA MEMÓRIA LOCAL A PARTIR DO ENDEREÇO 50H
36. ADDI R11,#01
37. ADDI R10,#01
38. XOR R12,R10,R9
39. JMPZD #FIM
40. JMPD #VOLTA2

41. FIM:
42. HALT
```

Figura 82 - Programa escrito na memória local do processador R8.

A segunda etapa do processo de execução corresponde à inicialização do processador, através do envio do comando de *reset* para o processador R8, pelo núcleo Serial_OCP.

Após a recepção do comando de *reset*, o processador executa o programa descrito acima.

Ao final da execução, o núcleo Serial_OCP realiza a leitura da memória local (RAM), a partir do endereço 50H, verificando a igualdade entre os valores lidos e os valores originais.

A Figura 83 apresenta o núcleo Serial_OCP enviando *flits* para a HERMES-OCP (rede-IP). Estes *flits* são direcionados ao núcleo R8_OCP, a Figura 83 apresenta apenas a recepção e envio dos *flits* da chave 10.

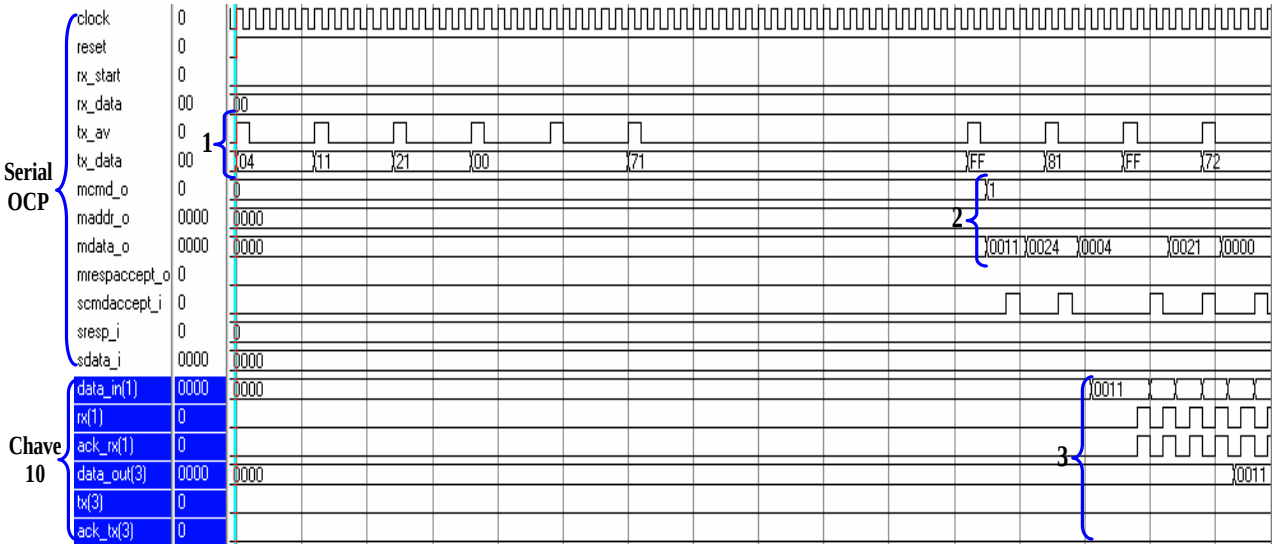


Figura 83 - Simulação do programa serial enviando flits na rede com destino ao núcleo R8_OCP conectado à chave 11 (endereço 0011).

1. O núcleo Serial OCP recebe os *flits* do programa serial. A existência de dados é indicada pelo sinal *tx_av* = "1" e pelos dados no sinal *tx_data*. São necessários 6 ciclos de relógio para recepção de cada dado (*flit*).

2. O núcleo Serial OCP inicia as transações de escrita OCP. Nesse instante, o sinal Mcmd_o recebe o valor "1" e o sinal Mdata_o recebe os *flits* que devem ser enviados a rede.

3. A chave 10 recebe os flits pela porta leste (índice 1). Após 10 ciclos de relógio a chave coloca o primeiro *flit* (correspondente a endereço de destino "0011") no sinal *data_out*(3) e ativa o sinal *tx*(3) da porta sul. Uma vez que o sinal *ack_tx*(3) está ativo o *flit* é removido do buffer e capturado do sinal data_in(2) da chave 11 (não representada na Figura 83).

Após alguns ciclos de relógio a IR-E (Escravo OCP) do núcleo R8_OCP recebe o pacote enviado pela Serial_OCP. A Figura 84 ilustra a recepção por parte do R8_OCP e a inserção dos dados (programa descrito na Figura 82) na memória local (RAM).

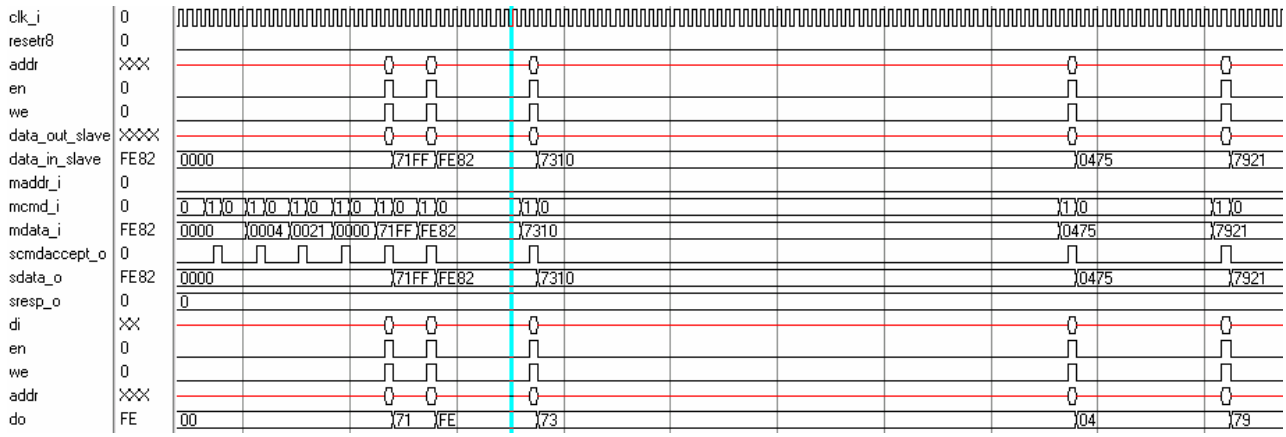


Figura 84 - Simulação da recepção dos *flits* pela R8 e inserção dos dados na memória local (RAM).

6.4 Resultados de Área Ocupada

O sistema em questão foi prototipado na plataforma de prototipação Memec Insight que contém um dispositivo FPGA Virtex-II 1000 de um milhão de portas lógicas num CI de 456 pinos. Além disso, a plataforma oferece também uma memória DDR (*Double Data Rate*) de 16M x 16 bits, uma PROM XC18V04 de 4MBits, duas fontes de relógio (uma de 24MHz e outra de 100MHz), uma porta RS-232 serial e conector de 160 pinos para permitir acrescentar módulos especiais. Junto à plataforma, existe uma interface LVDS (*Low-voltage Differential Signaling*) que provê transmissão de dados de 16 bits em alta velocidade [XIL02].

Para validar este sistema utilizou-se o ChipScope, da Xilinx. ChipScope permite a visualização dos sinais internos do FPGA selecionados pelo usuário em tempo de execução. A rede-IP com topologia malha de dimensão 2X2 (duas *BlockRAM* com IR-E OCP) foi sintetizada usando a ferramenta de síntese Leonardo Spectrum. A Tabela 11 apresenta os dados de área gerados por síntese, mostrando o uso dos recursos do FPGA e para ASIC.

Tabela 11 - Dados de área da NOC 2x2 com um módulo Serial, duas memórias *BlockRam* e um processador R8 todos com IRs OCP para XC2V1000 FPGA.

Modelo NORMA	FPGA		ASIC
	LUTs	Ocupação de LUTs na Virtex II	GATES*
NoC 2x2	1016	19,84%	16739
IR-M	99	1,93%	1273
IR-E	86	1,68%	1211
IR-ME	189	3,69%	2522
Módulo Serial OCP	195	3,81%	2267
Módulo Memória (*BlockRam* OCP)	53	1,04%	929
Módulo R8 OCP	485	9,47%	4953
Sistema	**2266**	**44,26%**	**33088**

* Utilizou-se a tecnologia TSMC 0.35 micron (typ)

7 CONCLUSÃO

O presente trabalho discutiu o problema de integração de núcleos em dois aspectos: (*i*) estruturas de interconexão e (*ii*) integração de núcleos com interfaces externas padrão. Com isso foram descritas as estruturas de interconexão utilizadas para interconexão de núcleo em SoCs com ênfase em redes intra-chip. Além de apresentar os conceitos básicos inerentes a essa abordagem de interconexão, relatou-se o estado da arte de redes intra-chip. No que tange a questão de interfaceamento de núcleos enfatizou-se o estudo do protocolo de comunicação padrão OCP.

A partir dos resultados obtidos acredita-se que o presente trabalho contribui para o avanço em duas áreas de pesquisa de relevância tanto no meio acadêmico como na indústria: interfaces padrão de comunicação e redes intra-chip. Dessa forma citam-se as contribuições deste trabalho: (*i*) redução de área da rede através da remoção das filas não utilizadas; (*ii*) parametrização da HERMES; (*iii*) desenvolvimento das interfaces de rede OCP (mestre, escravo, mestre-escravo) para a rede HERMES; (*iv*) unificação das contribuições citadas anteriormente através da ferramenta Maia; (*v*) descrição de um estudo de caso de desenvolvimento de SoC com a rede HERMES-OCP prototipada em FPGA.

Além das contribuições citadas acima, este trabalho trouxe ao grupo uma experiência no campo de protocolos de comunicação padrão, enfatizando o OCP. Dentro deste contexto, pode-se citar a certificação de núcleos com interface OCP frente ao fluxo da ferramenta *CoreCreator* como uma dificuldade encontrada no desenvolvimento do presente trabalho.

A prototipação em FPGA foi outra dificuldade encontrada. Mesmo havendo uma simulação funcional perfeita, podem ocorrer casos em que o sistema simplesmente não funciona no FPGA. Situações como não inicialização de máquinas de estado, não inicialização de registradores, atribuições onde não se consideram todos os casos possíveis (ocasionando o *warning latch infered*), freqüência do relógio da placa superior à freqüência de operação do circuito, por exemplo, são problemas difíceis de detectar e ocasionam atrasos no desenvolvimento do projeto. A fase de prototipação é imprescindível a qualquer projeto de sistemas computacionais, pois prova que o sistema realmente funciona no hardware e não é apenas um conjunto de formas de onda em um simulador.

Como trabalhos futuros citam-se: (*i*) estudar técnicas para suporte a QoS, e garantir entrega ordenada de dados no caso de concorrência; (*ii*) ampliar o número de comandos suportados pelas IRs OCP; (*iii*) inserir na ferramenta Maia a possibilidade de gerar trafego real (e.g. trafego *streaming*) para aplicações de SoCs, como decodificação de vídeo ou imagem; (*iv*) pesquisar e implementar novos algoritmos de roteamento que sejam imunes a *deadlock*.

8 REFERÊNCIAS BIBLIOGRÁFICAS

[ALT02] ALTERA. **Avalon Bus Specification. Reference Manual**. Document Version 1.2, 2002.

[AMO02] Amory, A.M. **Integração e Avaliação de Técnicas de Teste Baseado em Software no Fluxo de Projeto de SOCs**. Dissertação de mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2002, 133 p.

[AMO03] Amory, A. M. **MET2 – A microprocessor for Embedded Test - Designer's Manual**. Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação, 2003, 27 p.

[AND03] Andriahantenaina, A.; Greiner, A. **Micro-network for SoC: Implementation of a 32-port SPIN network**. In: Design Automation and Test in Europe (DATE'03), 2003, pp. 1128 –1129.

[AND03a] Andriahantenaina, A.; Charlery, H.; Greiner, A.; Mortiez, L.; Zeferino C. **SPIN: a Scalable, Packet Switched, On-chip Micro-network**. In: Design Automation and Test in Europe (DATE'03), 2003, pp. 70 –73.

[ARM02] ARM Corp. **AMBA 2.0 Specification**. Capturado em: http://www.arm.com/armtech/AMBA_Spec, 2000.

[BEN01] Benini, L.; De Micheli,G. **Powering Networks on Chip**. In: International Symposium on System Synthesis, 2001, pp. 33 –38.

[BEN02] Benini, L.; De Micheli,G. **Networks on chips: a new SoC paradigm**. Computer, v. 35(**1**), January 2002, pp. 70-78.

[BER00] Bergamaschi, R. A.; Lee, W. R. **Designing systens-on-chip using cores**. In: Design Automation Conference (DAC'00), 2000, pp. 420-425.

[BER01] Bergamaschi, R. A.; Bhattacharya, S.; Wagner, R.; Fellenz, C.; Muhlada, M.; White, F.; Daveau, J. M.; Lee, W. R. **Automating the design of SoCs using cores.** IEEE Design & Test of Computers, v. 18(**5**), Sep.-Oct. 2001, pp. 32 -45.

[BER02] Bergamaschi, R.A.; Cohn, J. **The A to Z of SoCs**. In: International Conference on Computer Aided Design (ICCAD), 2002, pp. 791-798.

[BIR99] Birnbaum, M.; Sahs, H. **How VSIA Answers the SoC Dilemma**. Computer, v. 32(**6**), June 1999, pp. 42-50.

[DAL00] Dallpasso, M.; Bogliolo, A., Benini, L. **Hardware/software IP protection**. In: Design Automation Conference (DAC'00), 2000, pp. 593-596.

[DAL01] Dally, W.; Towles, B. **Route packets, not wires: on-chip interconnection networks**. In: Design Automation Conference (DAC'01), 2001, pp. 684-689.

[DAY83] Day, J.; Zimmermman, H. **The OSI reference model**. Proceedings of the IEEE, v. 71(**12**), Dec. 1983, pp. 1334-1340.

[DUA97] Duato, J. *et al*. **Interconnection Networks**. IEEE Computer Society Press, Los Alamitos California, 1997, 515 p.

[FOR02] Forsell, M. **A Scalable High-Performance Computing Solution for Networks on Chips**. IEEE Micro, Sep.-Oct. 2002, v. 22(**5**), pp. 46-55.

[GAJ00] Gajski,D., Zhu,J., Dömer,R., Gerstlauer,A., Zhao,S. **SpecC: Specification Language and Methodology**. Kluwer Academic Publishers, Norwell, MA, 2000, 336 p.

[GLA94] Glass, C.; Ni, L. **The Turn Model for Adaptive Routing.** Journal of the Association for Computing Machinery, v. 41(**5**), Sep. 1994, pp. 874-902.

[GOO03] Goossens, K. et al. **Guaranteeing the Quality of Services in Network on a Chip.** In: Axel Jantsch and Hannu Tenhunen, editors, *Networks on Chip*, chapter 4, Kluwer Academic Publishers, 2003, pp. 61-82.

[GUE00] Guerrier. P.; Greiner. A. **A generic architecture for on-chip packet-switched interconections**. In: Design Automation and Test in Europe (DATE'00), 2000, pp. 250-256.

[GUE99] Guerrier. P.; Greiner. A. **A Scalable Architecture for System-On-Chip Interconnections**. In: Sophia Antipolis Forum on Microelectronics (SAME), 1999, pp. 90-93.

[GUP97] Gupta, R. K., Zorian, Y. **Introducing Core-Based System Design**. IEEE Design & Test of Computers, v.14(**4**), Oct.-Dec. 1997, pp. 15-25.

[HWA93] Hwang, K. **Advanced Computer Architecture: Parallelism, Scalability, Programmability**. New York: McGraw-Hill, 1993, 771p.

[IBM02] IBM. **The CoreConnectTM Bus Architecture**. Capturado em: http://www.3ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture, 2002.

[IEE02] **IEEE P1500 Web Site**. Capturado em: http://grouper.ieee.org/groups/1500/, 2002.

[IEE02a] IEEE Standards Association. IEEE Standard Test Access Port and Boundary-Scan Architecture (1149.1). Capturado em: http://grouper.ieee.org/groups/1149/, 2002.

[ITR02] International Sematech. **International Technology Roadmap for Semiconductors – 2002 Update**, Capturado em: http://public.itrs.net, 2002.

[JUN01] Juneidi, Z.; Torki, K.; Martinez, S.; Nicolescu, G.; Courtois, B.; Jerraya, A. **Global Modeling and Simulation of System-on-Chip embedding MEMS devices**. In: International Conference on ASIC, 2001, pp. 666 –669.

[KAR01] Karim, F.; Nguyen, A.; Dey, S.; Rao, R. **On-chip communication architecture for OC-768 network processors.** In: Design Automation Conference (DAC'01), 2001, pp. 678-683.

[KAR02] Karim, F.; Nguyen, A.; Dey S. **An interconnect architecture for network systems on chips**. IEEE Micro v. 22(**5**), Sep.-Oct. 2002, pp. 36-45.

[KUM02] Kumar, S.; Jantsch, A.; Soininen, J. P.; Fonsell, M. **A Network on Chip Architecture and Design Methodology**. In: Computer Society Annual Symposium on VLSI (ISVLSI'02), 2002, pp. 105-112.

[KUM03] Kumar, S. **On Packet Switched Network for Chip Communication.** In: Axel Jantsch and Hannu Tenhunen, editors, *Networks on Chip*, chapter 5, Kluwer Academic Publishers, 2003, pp. 85-106.

[LIA00] Liang, J.; Swaminathan, S.; Tessier, R. **aSOC: A Scalable, Single-Chip communications Architecture**. In: IEEE International Conference on Parallel Architectures and Compilation Techniques, 2000, pp. 37-46.

[MAD97] Madisetti, V. K., Shen L. **Interface Design for Core-Based Systems**. IEEE Design & Test of Computers, v. 14(**4**), October-December 1997, pp. 45-51.

[MAR01] Martin, G.; Chang, H. **System on Chip Design**. In: International Symposium on Integrated Circuits, Devices & Systems (ISIC'01), Tutorial 2, 2001.

[MAR02] Marescaux, T.; Bartic, A.; Verkest, D.; Vernalde, S.; Lauwereins, R. **Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs**. In: Field-Programmable Logic and Applications (FPL'02), 2002, pp. 795-805.

[MEL03] Mello, A. V. e Möller, L.H. **Arquitetura Multiprocessada em SoCs: Estudo de Diferentes Topologias de Conexão.** Trabalho de Conclusão, FACIN-PUCRS, Julho, 2003. Capturado em: http://www.inf.pucrs.br/~moraes/papers/tc_multiproc.pdf, 120p.

[MOD02] ModelSim EE/PLUS User's Manual, chapter 11 – **VHDL Foreign Language Interface and Verilog**.

[MOR03] Moraes, F. G.; Mello, A.; Möller, L.; Ost, L.; Calazans, N. **A Low Area Overhead Packet-switched Network on Chip: Architecture and Prototyping**. In: Very Large Scale Integration (VLSI-SOC), 2003, pp. 318-323.

[MOR03a] Moraes, F. G.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. **HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip**. Integration VLSI Journal, 2003 (aceito para publicação).

[MOR03b] Moreno, E. **Modelando, descrevendo e validando NoCs para SoCs em diferentes níveis de abstração**. Dissertação de mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2003.

[NI93] Ni, L. M.; McKinley, P. K. **A Survey of Wormhole Routing Techniques in Direct Networks**. IEEE Computer Magazine, v.26(**2**), February, 1993, pp. 62-76.

[OCP02] OCP-IP. **Open Core Protocol Specification**. Versão 1.0. Capturado em: http://www.ocpip.org/, 2002.

[OCP02a] OCP-IP. White Paper - **The Importance of Sockets in SoC Design**. Capturado em: http://www.ocpip.org/socket/whitep/, 2002.

[OCP02b] OCP-IP. **OCP-IP News – The Official Newsletter of the Open Core Protocol International Partnership**, Capturado em: http://www.ocpip.org/data/pressroom/newsletter/OCP_Newsletter_10-2002.pdf, 2002.

[OCP02c] OCP-IP. White Paper - **Socket-Centric IP Core Interface Maximizes IP Applications**. Capturado em: http://www.ocpip.org/socket/whitep/, 2002.

[OCP03] OCP-IP. **Open Core Protocol Specification**. Versão 2.0. Capturado em: http://www.ocpip.org/, 2003.

[PAN03] Pande, P.; Grecu, C.; Ivanov, A.; Saleh, R. **Design of a switch for network on chip applications**. In: International Symposium on Circuits and Systems (ISCAS'03), 2003, pp. 217-220.

[PEH00] Peh, Li-Shiuan; Dally, W.J. **Flit – Reservation Flow Control**. High-Performance Computer Architecture, 2000, pp. 73 –84.

[PER98] Perry, Douglas. **VHDL 3rd Edition**, McGraw-Hill, New York, US, 1998, 493 p.

[PRA03] Praven, B.; Mahapatra, R. **Interfacing cores with On-chip Packet-Switched Networks.** In: Very Large Scale Integration (VLSI), 2003, pp. 382-387.

[REI00] Reis, R. A. L.; **Concepção de Circuitos Integrados**. Instituto de Informática da UFRGS. Editora Sagra Luzzatto, 2000, 252 p.

[RIJ03] Rijpkema, E.; et. al. **A Router Architecture for Network on Silicon.** In: Workshop on Embedded Systems (PROGRESS´2001), 2001, pp. 181-188.

[RIJ03a] Rijpkema, E.; et. al. **Trade Offs in the Design of a Router with both Guaranteed and Best-Effort Services for Networks On Chip.** In: Design Automation Test Europe (DATE'03), 2003, pp. 350-355.

[RUN00] Runner, S.; Sanaka, V.; Yu, E. **Building an Infrastructure for IP Reuse**. EE Times, May 15, 2000. Capturado em: www.eetimes.com, 2002.

[SAA02] Saastamoinen, I.; et. al. **Interconnect IP Node for Future System-on-Chip Designs**. In: Workshop on Electronic Design, Test and Applications (DELTA'02), 2002, pp.116–120.

[SAA02a] Saastamoinen, I.; Alho, M.; Pirttimäki, J.; Nurmi, J. **Proteo Interconnect IPs for Networks-on-Chip**. In: IP Based SoC Design, 2002.

[SAA03] Saastamoinen, I; Alho, M.; Nurmi, J. **Buffer Implementation for Proteo Networks-on-Chip**. In: International Symposium on Circuits and Systems (ISCAS'03), 2003, pp. 113-116.

[SGR01] Sgroi, M.; Sheets, M.; Mihal, A.; Keutzer, K.; Malik, S.; Rabaey, J.; Vincentelli, A. S. **Addressing the System-on-Chip Interconnect Woes Through Communication-Based Design**. In: Design Automation Conference (DAC'01), 2001, pp. 667 –672.

[SIA99] SIA - Semiconductor Industry Association, The National Technology Roadmap for Semiconductors 1999. Capturado em: http://public.itrs.net/, Julho 2002.

[SIG02] Sigüenza-Tortosa, D.; Nurmi, J. **Proteo: A New Approach to Network-on-Chip.** In: IASTED International Conference on Communication Systems and Networks (CSN'02), 2002.

[SON03] **CoreCreator Overview**. Capturado em: http://www.sonicsinc.com/sonics/products/siliconbackplane/corecreator.html, 2003.

[SYS03] **SystemC v2.0.1 Language Reference Manual**, Capturado em: http://www.systemc.org/projects/systemc/document/SystemC_v201_LRM/, 2003.

[THO91] Thomas, D., Moorby, P. **The Verilog Hardware Description Language**. Kluwer Academic Publishers, 1991, p 376.

[VSI02] VSI Alliance. Capturado em: http://www.vsi.org/.

[WIL03] Wilson, R. **OCP, VCI join forces for SoC interconnect**. EE Times, October 7, 2003. Capturado em: www.eetimes.com, 2003.

[XIL02] Xilinx, Inc. **VIRTEX-II User Guide**, 2002.

[YE03] Ye, T.; Benini, L.; De Micheli, G. **Packetized On-Chip Interconnection Communication Analysis for MPSoC**. In: Design Automation and Test in Europe (DATE'03), 2003, pp. 344-349.

[ZEF03] Zeferino, C. A. and Susin, A. A. "**SoCIN: A Parametric and Scalable Network-on-Chip**", In: Symposium on Integrated Circuits and Systems (SBCCI'2003), 2003, pp. 169-174.

[ZEF03a] Zeferino, C. A. "**Redes-em-chip: Arquiteturas e Modelos para Avaliação de Área e Desempenho**". Tese de doutorado, Universidade Federal do Rio Grande do Sul - UFRGS/PPGC, Porto Alegre, RS, 2003.

ANEXO I – MÁQUINAS DE ESTADO

Este Anexo apresenta as máquinas de estado de controle responsáveis pela interface entre o protocolo OCP e os núcleos IP *BlockRAM* e *Serial*.

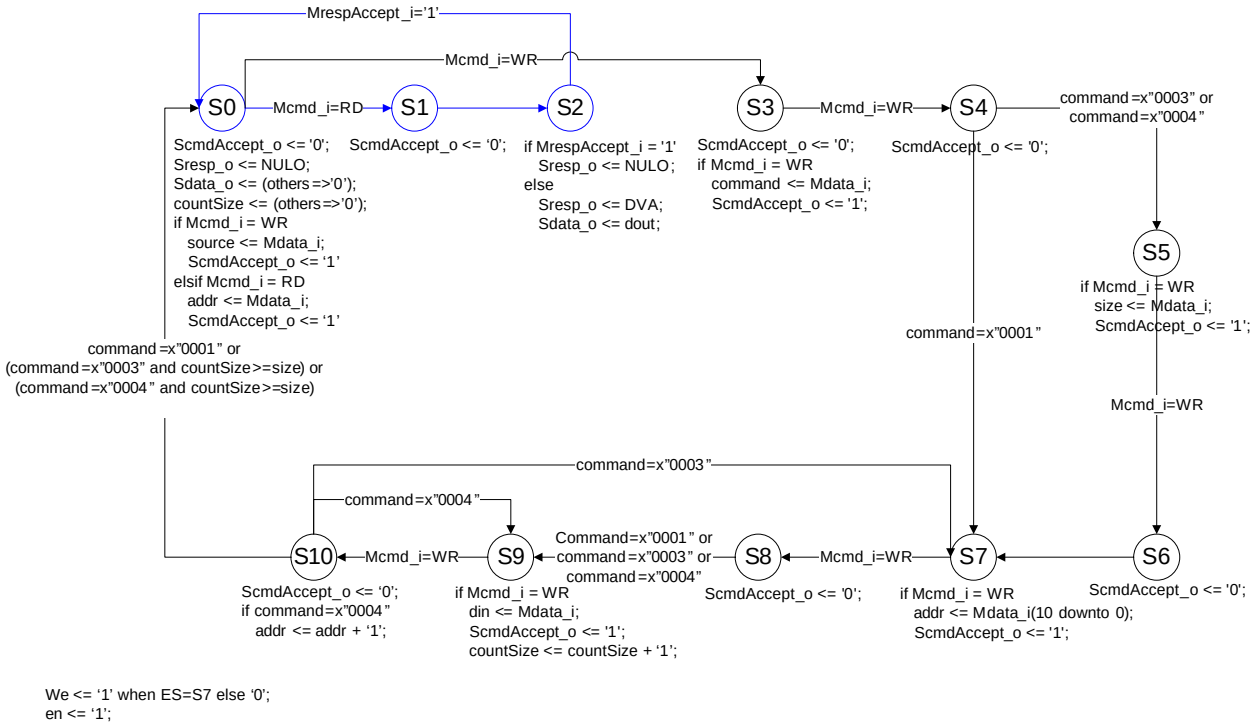


Figura 85 - Máquina de estados que implementa os comando da memória (BlockRAM).

Os estados referentes a cada um dos comandos descritos no Capítulo 6, módulo BlockRAM:

- (S0, S1 e S2) referentes ao comando Read Simples.
- (S0, S3, S4, S7, S8, S9, S10) utilizados para uma escrita com comando Write Simples.
- (S0, S3, S4, S5, S6, S7, S8, S9, S10) utilizados para escrita com o comando Write Burst modo 1.
- (S0, S3, S4, S5, S6, S7, S8, S9, S10) utilizados para escrita com o comando Write Burst modo 2.

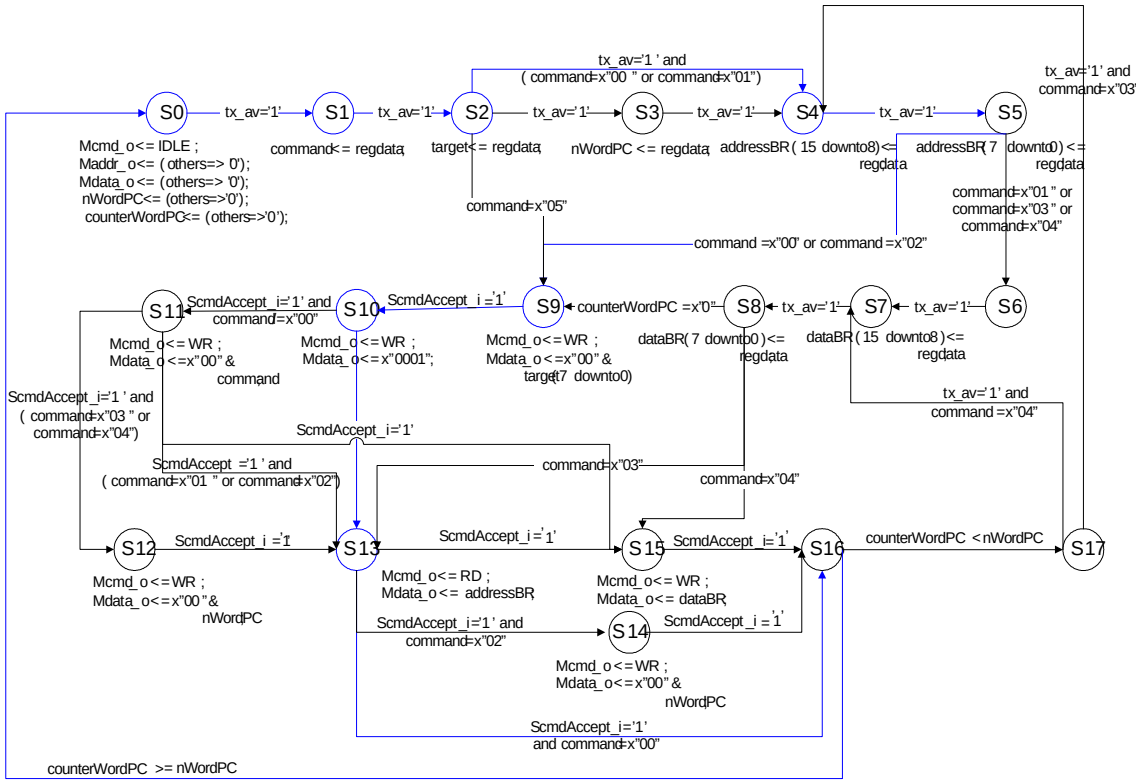


Figura 86 - Máquina de estados do módulo Serial.

Os estados referentes a cada um dos comandos descritos no Capítulo 6, módulo Serial:

- (S0, S1, S2, S4, S9, S10, S13, S16) referentes ao comando Read Simples .
- (S0, S1, S2, S4, S5, S6, S7, S8, S9, S10, S11, S13, S15, S16) referentes ao comando Write Simples.
- (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S15, S16, S17) referentes ao comando Write Burst modo 1.
- (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S15, S16, S17) referentes ao comando Write Burst modo 2.
- (S0, S1, S2, S3, S4, S9, S10, S11, S15, S16) referentes ao comando Reset.

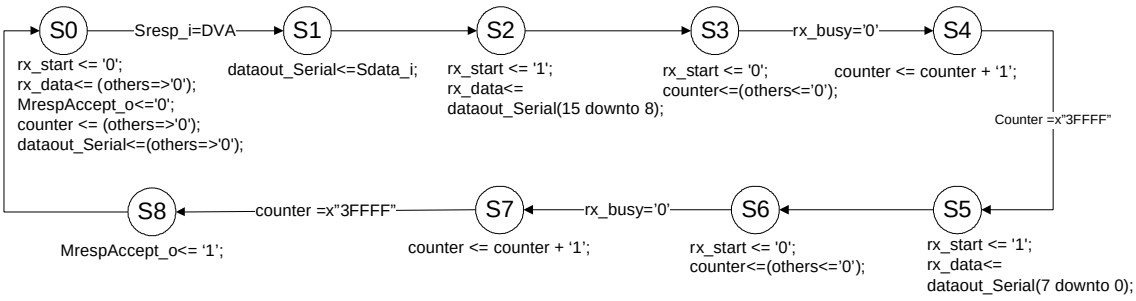


Figura 87 - Máquina de estados para recepção de dados pelo módulo Serial.

ANEXO II – HIERARQUIA DOS VHDLS DE UMA REDE-IP

A Figura 88 ilustra a hierarquia dos arquivos VHDLs que compõe os pares entidade-arquitetura do sistema R8-OCP cuja estrutura de interconexão é uma rede-IP com IRs OCP gerada pela ferramenta Maia.

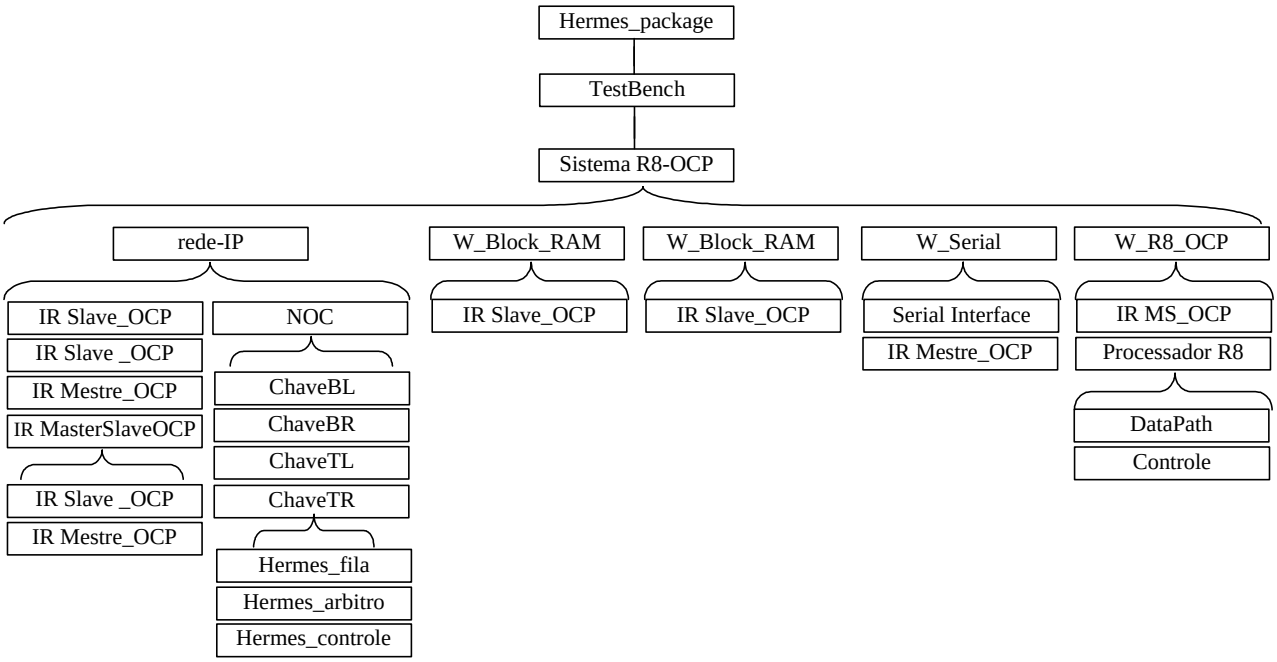


Figura 88 - Hierarquia dos pares entidade-arquiterura de um SoC com uma rede-IP OCP.

Os códigos VHDLs referentes aos pares entidade-arquitetura de uma rede-IP com IRs OCP são apresentados na seguinte ordem: (*i*) Anexo III – Hermes_package.vhd; (*ii*) Anexo IV – NoCOCP.vhd; (*iii*) Anexo V – Slave_OCP.vhd; (*iv*) Anexo VI – Master_OCP.vhd; (*v*) Anexo VII – MasterSlave_OCP.vhd; (*vi*) Anexo VIII - NOC.vhd.

ANEXO III – CÓDIGO VHDL HERMES_PACKAGE.VHD

```
--------------------------------------------------------------------------------
-- package com tipos basicos
--------------------------------------------------------------------------------
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.std_logic_unsigned.all;

package HermesPackage is

------------------------------------------------------------------------
-- OCP PARAMETERS
------------------------------------------------------------------------
------------------ command Enconding - p. 13 ---------------------------
       constant IDLE: Std_Logic_Vector(2 downto 0) :="000";
       constant WR:   Std_Logic_Vector(2 downto 0) :="001";
       constant RD:   Std_Logic_Vector(2 downto 0) :="010";
       constant RDEX: Std_Logic_Vector(2 downto 0) :="011";
       constant BCST: Std_Logic_Vector(2 downto 0) :="111";
----------------------Response Enconding-----------------------------
       constant DVA:  Std_Logic_Vector(1 downto 0) :="01";
       constant ERR:  Std_Logic_Vector(1 downto 0) :="11";
       constant NULO: Std_Logic_Vector(1 downto 0) :="00";
       constant ALVO: Std_Logic_Vector(7 downto 0) :="00000000";
------------------------------------------------------------
-- CONSTANTS INDEPENDENTES
------------------------------------------------------------
      constant NPORT: integer := 5;

      constant EAST  : integer := 0;
      constant WEST  : integer := 1;
      constant NORTH : integer := 2;
      constant SOUTH : integer := 3;
      constant LOCAL : integer := 4;
------------------------------------------------------------
-- CONSTANT DEPENDENTE DA LARGURA DE BANDA DA REDE
------------------------------------------------------------
      constant TAM_FLIT : integer range 1 to 32 := 16;
      constant METADEFLIT : integer range 1 to 16 := (TAM_FLIT/2);
      constant QUARTOFLIT : integer range 1 to 16 := (TAM_FLIT/4);
------------------------------------------------------------
-- CONSTANTS DEPENDENTES DA PROFUNDIDADE DA FILA
------------------------------------------------------------
      constant TAM_BUFFER: integer := 8;
      constant TAM_POINTER : integer range 1 to 32 := 3;
------------------------------------------------------------
-- CONSTANTS DEPENDENTES DO NUMERO DE CHAVES
------------------------------------------------------------
      constant NROT: integer := 4;
      constant MIN_X : integer := 0;
      constant MIN_Y : integer := 0;
      constant MAX_X : integer := 1;
      constant MAX_Y : integer := 1;
------------------------------------------------------------
-- CONSTANT TB
------------------------------------------------------------
      constant TAM_LINHA : integer := 4;
      constant N0000: integer :=0;
      constant ADDRESSN0000: std_logic_vector(7 downto 0) :="00000000";
      constant N0100: integer :=1;
      constant ADDRESSN0100: std_logic_vector(7 downto 0) :="00010000";
      constant N0001: integer :=2;
      constant ADDRESSN0001: std_logic_vector(7 downto 0) :="00000001";
      constant N0101: integer :=3;
      constant ADDRESSN0101: std_logic_vector(7 downto 0) :="00010001";
------------------------------------------------------------
-- SUBTIPOS, TIPOS E FUNCOES
------------------------------------------------------------
      subtype reg3 is std_logic_vector(2 downto 0);
      subtype reg8 is std_logic_vector(7 downto 0);
      subtype regNrot is std_logic_vector((NROT-1) downto 0);
      subtype regNport is std_logic_vector((NPORT-1) downto 0);
      subtype regflit is std_logic_vector((TAM_FLIT-1) downto 0);
      subtype regmetadeflit is std_logic_vector(((TAM_FLIT/2)-1) downto 0);
      subtype regquartoflit is std_logic_vector((QUARTOFLIT-1) downto 0);
      subtype pointer is std_logic_vector((TAM_POINTER-1) downto 0);
```

```
      type buff is array(0 to TAM_BUFFER-1) of regflit;
      type arrayNport_reg3 is array((NPORT-1) downto 0) of reg3;
      type arrayNport_reg8 is array((NPORT-1) downto 0) of reg8;
      type arrayNport_regflit is array((NPORT-1) downto 0) of regflit;
      type arrayNrot_regflit is array((NROT-1) downto 0) of regflit;
      type arrayNrot_regmetadeflit is array((NROT-1) downto 0) of regmetadeflit;
      function CONV_VECTOR( int: integer ) return std_logic_vector;

-------------------------------------------------------
-- FUNCOES TB
-------------------------------------------------------
      function CONV_VECTOR( letra : string(1 to TAM_LINHA);  pos: integer ) return std_logic_vector;
      function CONV_HEX( int : integer ) return string;
      function CONV_STRING_4BITS( dado : std_logic_vector(3 downto 0)) return string;
      function CONV_STRING_8BITS( dado : std_logic_vector(7 downto 0)) return string;
      function CONV_STRING_16BITS( dado : std_logic_vector(15 downto 0)) return string;
      function CONV_STRING_32BITS( dado : std_logic_vector(31 downto 0)) return string;

end HermesPackage;
package body HermesPackage is
-------------------------------------------------------
-- converte um inteiro em um std_logic_vector(2 downto 0)
-- -----------------------------------------------------
function CONV_VECTOR( int: integer ) return std_logic_vector is
      variable bin: reg3;
          begin
              case(int) is
                          when 0 => bin := "000";
                          when 1 => bin := "001";
                          when 2 => bin := "010";
                          when 3 => bin := "011";
                          when 4 => bin := "100";
                          when 5 => bin := "101";
                          when 6 => bin := "110";
                          when 7 => bin := "111";
                          when others => bin := "000";
              end case;
          return bin;
end CONV_VECTOR;
-------------------------------------------------------
-- FUNCOES TB
-------------------------------------------------------
    -- ###########converte um caracter de uma dada linha em um std_logic_vector#########

  function CONV_VECTOR( letra:string(1 to TAM_LINHA);  pos: integer ) return std_logic_vector is
     variable bin: std_logic_vector(3 downto 0);
   begin
      case (letra(pos)) is
              when '0' => bin := "0000";
              when '1' => bin := "0001";
              when '2' => bin := "0010";
              when '3' => bin := "0011";
              when '4' => bin := "0100";
              when '5' => bin := "0101";
              when '6' => bin := "0110";
              when '7' => bin := "0111";
              when '8' => bin := "1000";
              when '9' => bin := "1001";
              when 'A' => bin := "1010";
              when 'B' => bin := "1011";
              when 'C' => bin := "1100";
              when 'D' => bin := "1101";
              when 'E' => bin := "1110";
              when 'F' => bin := "1111";
              when others =>  bin := "0000";
      end case;
     return bin;
  end CONV_VECTOR;

-- converte um inteiro em um string
      function CONV_HEX( int: integer ) return string is
            variable str: string(1 to 1);
      begin
            case(int) is
                  when 0 => str := "0";
                  when 1 => str := "1";
                  when 2 => str := "2";
                  when 3 => str := "3";
                  when 4 => str := "4";
                  when 5 => str := "5";
```

```
                when 6 => str := "6";
                when 7 => str := "7";
                when 8 => str := "8";
                when 9 => str := "9";
                when 10 => str := "A";
                when 11 => str := "B";
                when 12 => str := "C";
                when 13 => str := "D";
                when 14 => str := "E";
                when 15 => str := "F";
                when others =>  str := "U";
          end case;
          return str;
    end CONV_HEX;

    function CONV_STRING_4BITS(dado : std_logic_vector(3 downto 0)) return string is
          variable str: string(1 to 1);
    begin
          str := CONV_HEX(CONV_INTEGER(dado));
          return str;
    end CONV_STRING_4BITS;

    function CONV_STRING_8BITS(dado : std_logic_vector(7 downto 0)) return string is
          variable str1,str2: string(1 to 1);
          variable str: string(1 to 2);
    begin
          str1 := CONV_STRING_4BITS(dado(7 downto 4));
          str2 := CONV_STRING_4BITS(dado(3 downto 0));
          str := str1 & str2;
          return str;
    end CONV_STRING_8BITS;

    function CONV_STRING_16BITS(dado : std_logic_vector(15 downto 0)) return string is
          variable str1,str2: string(1 to 2);
          variable str: string(1 to 4);
    begin
          str1 := CONV_STRING_8BITS(dado(15 downto 8));
          str2 := CONV_STRING_8BITS(dado(7 downto 0));
          str := str1 & str2;
          return str;
    end CONV_STRING_16BITS;

    function CONV_STRING_32BITS(dado : std_logic_vector(31 downto 0)) return string is
          variable str1,str2: string(1 to 4);
          variable str: string(1 to 8);
    begin
          str1 := CONV_STRING_16BITS(dado(31 downto 16));
          str2 := CONV_STRING_16BITS(dado(15 downto 0));
          str := str1 & str2;
          return str;
    end CONV_STRING_32BITS;

end HermesPackage;
```

ANEXO IV – CÓDIGO VHDL NOCOCP.VHD

O código VHDL do par entidade-arquitetura de uma rede-IP de dimensão 2x2:

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.HermesPackage.all;

entity NoCOCP is
port(
      clock:             in std_logic;
      reset:             in std_logic;
      ----------------SLAVE SIGNALS NODO 00-------------------
      N0000_Sclk_i:        in  std_logic;
      N0000_Sreset_ni:     in  std_logic;
      N0000_Scmd_i:        in  reg3;                          -- ocp command
      N0000_Saddr_i:       in  regflit;                       -- ocp address
      N0000_Sdata_i:       in  regflit;                       -- ocp write data
      N0000_ScmdAccept_o:  out std_logic;                     -- ocp command accept
      N0000_Sresp_o:       out std_logic_vector(1 downto 0); -- ocp response
      N0000_Sdata_o:       out regflit;                       -- ocp read data
      N0000_SrespAccept_i: in  std_logic;
      ---------------MASTER SIGNALS NODO 10------------------
      N0100_Mclk_i:        in  std_logic;
      N0100_Mreset_ni:     in  std_logic;
      N0100_Mcmd_o:        out reg3;                          -- ocp command
      N0100_Maddr_o:       out regflit;                       -- ocp address
      N0100_Mdata_o:       out regflit;                       -- ocp write data
      N0100_McmdAccept_i:  in  std_logic;                     -- ocp command accept
      N0100_Mresp_i:       in  std_logic_vector(1 downto 0); -- ocp response
      N0100_Mdata_i:       in  regflit;                       -- ocp read data
      N0100_MrespAccept_o: out std_logic;
      ---------------MASTER SIGNALS NODO 01------------------
      N0001_Mclk_i:        in  std_logic;
      N0001_Mreset_ni:     in  std_logic;
      N0001_Mcmd_o:        out reg3;                          -- ocp command
      N0001_Maddr_o:       out regflit;                       -- ocp address
      N0001_Mdata_o:       out regflit;                       -- ocp write data
      N0001_McmdAccept_i:  in  std_logic;                     -- ocp command accept
      N0001_Mresp_i:       in  std_logic_vector(1 downto 0); -- ocp response
      N0001_Mdata_i:       in  regflit;                       -- ocp read data
      N0001_MrespAccept_o: out std_logic;
      -------------MASTERSLAVE SIGNALS NODO 11---------------
      N0101_Mclk_i:        in  std_logic;
      N0101_Mreset_ni:     in  std_logic;
      N0101_Mcmd_o:        out reg3;                          -- ocp command
      N0101_Maddr_o:       out regflit;                       -- ocp address
      N0101_Mdata_o:       out regflit;                       -- ocp write data
      N0101_McmdAccept_i:  in  std_logic;                     -- ocp command accept
      N0101_Mresp_i:       in  std_logic_vector(1 downto 0); -- ocp response
      N0101_Mdata_i:       in  regflit;                       -- ocp read data
      N0101_MrespAccept_o: out std_logic;
      N0101_Sclk_i:        in  std_logic;
      N0101_Sreset_ni:     in  std_logic;
      N0101_Scmd_i:        in  reg3;                          -- ocp command
      N0101_Saddr_i:       in  regflit;                       -- ocp address
      N0101_Sdata_i:       in  regflit;                       -- ocp write data
      N0101_ScmdAccept_o:  out std_logic;                     -- ocp command accept
      N0101_Sresp_o:       out std_logic_vector(1 downto 0); -- ocp response
      N0101_Sdata_o:       out regflit;                       -- ocp read data
      N0101_SrespAccept_i: in  std_logic);
end NoCOCP;

architecture NoCOCP of NoCOCP is

------ SWITCHES SIGNALS --------
signal address : arrayNrot_regmetadeflit;
signal data_in, data_out : arrayNrot_regflit;
signal rx, ack_rx, tx, ack_tx : regNrot;

begin
      NOC: Entity work.NOC(NOC)
      port map(
            clock        => clock,
            reset        => reset,
```

```
      rxLocal      => rx,
      data_inLocal => data_in,
      ack_rxLocal  => ack_rx,
      txLocal      => tx,
      data_outLocal=> data_out,
      ack_txLocal  => ack_tx);

slave0000: Entity work.Slave_OCP(Slave_OCP)
port map(
      ----------- Slave OCP Signals------------------
      Sclk_i        => N0000_Sclk_i,
      Sreset_ni     => N0000_Sreset_ni,
      Scmd_i        => N0000_Scmd_i,
      Saddr_i       => N0000_Saddr_i,
      Sdata_i       => N0000_Sdata_i,
      ScmdAccept_o  => N0000_ScmdAccept_o,
      Sresp_o       => N0000_Sresp_o,
      Sdata_o       => N0000_Sdata_o,
      SrespAccept_i => N0000_SrespAccept_i,
      busyMaster    => '0',
      ----------- Switch Signals---------------------
      addressL      => addressN0000,
      rxL           => tx(N0000),
      data_inL      => data_out(N0000),
      ack_rxL       => ack_tx(N0000),
      txL           => rx(N0000),
      data_outL     => data_in(N0000),
      ack_txL       => ack_rx(N0000));

master0100: Entity work.Master_OCP(Master_OCP)
port map(
      ----------- Master OCP Signals------------------
      Mclk_i        => N0100_Mclk_i,
      Mreset_ni     => N0100_Mreset_ni,
      Mcmd_o        => N0100_Mcmd_o,
      Maddr_o       => N0100_Maddr_o,
      Mdata_o       => N0100_Mdata_o,
      McmdAccept_i  => N0100_McmdAccept_i,
      Mresp_i       => N0100_Mresp_i,
      Mdata_i       => N0100_Mdata_i,
      MrespAccept_o => N0100_MrespAccept_o,
      busySlave     => '0',
      ----------- Switch Signals---------------------
      rxL           => tx(N0100),
      data_inL      => data_out(N0100),
      ack_rxL       => ack_tx(N0100),
      txL           => rx(N0100),
      data_outL     => data_in(N0100),
      ack_txL       => ack_rx(N0100));

master0001: Entity work.Master_OCP(Master_OCP)
port map(
      ----------- Master OCP Signals------------------
      Mclk_i        => N0001_Mclk_i,
      Mreset_ni     => N0001_Mreset_ni,
      Mcmd_o        => N0001_Mcmd_o,
      Maddr_o       => N0001_Maddr_o,
      Mdata_o       => N0001_Mdata_o,
      McmdAccept_i  => N0001_McmdAccept_i,
      Mresp_i       => N0001_Mresp_i,
      Mdata_i       => N0001_Mdata_i,
      MrespAccept_o => N0001_MrespAccept_o,
      busySlave     => '0',
      ----------- Switch Signals---------------------
      rxL           => tx(N0001),
      data_inL      => data_out(N0001),
      ack_rxL       => ack_tx(N0001),
      txL           => rx(N0001),
      data_outL     => data_in(N0001),
      ack_txL       => ack_rx(N0001));

masterslave0101: Entity work.MasterSlave_OCP(MasterSlave_OCP)
port map(
      ----------- Master OCP Signals------------------
      Mclk_i        => N0101_Mclk_i,
      Mreset_ni     => N0101_Mreset_ni,
      Mcmd_o        => N0101_Mcmd_o,
      Maddr_o       => N0101_Maddr_o,
      Mdata_o       => N0101_Mdata_o,
      McmdAccept_i  => N0101_McmdAccept_i,
```

```
            Mresp_i       => N0101_Mresp_i,
            Mdata_i       => N0101_Mdata_i,
            MrespAccept_o => N0101_MrespAccept_o,
            ----------- Slave OCP Signals-------------------
            Sclk_i        => N0101_Sclk_i,
            Sreset_ni     => N0101_Sreset_ni,
            Scmd_i        => N0101_Scmd_i,
            Saddr_i       => N0101_Saddr_i,
            Sdata_i       => N0101_Sdata_i,
            ScmdAccept_o  => N0101_ScmdAccept_o,
            Sresp_o       => N0101_Sresp_o,
            Sdata_o       => N0101_Sdata_o,
            SrespAccept_i => N0101_SrespAccept_i,
            ----------- Switch Signals--------------------
            addressL      => addressN0101,
            rxL           => tx(N0101),
            data_inL      => data_out(N0101),
            ack_rxL       => ack_tx(N0101),
            txL           => rx(N0101),
            data_outL     => data_in(N0101),
            ack_txL       => ack_rx(N0101));

end NoCOCP;
```

ANEXO V – CÓDIGO VHDL MASTER_OCP.VHD

O código VHDL do par entidade-arquitetura da IR Master-OCP:

```
library IEEE;
use ieee.STD_LOGIC_UNSIGNED.all;
use ieee.std_logic_1164.all;
use work.HermesPackage.all;

--Master_OCP definition
entity Master_OCP is
port(
      ------------------------------OCP Signals-----------------------------------------------------
      Mclk_i:        in  std_logic;
      Mreset_ni:     in  std_logic;
      Mcmd_o:        out reg3;                        -- ocp command
      Maddr_o:       out std_logic;                   -- ocp address
      Mdata_o:       out regflit;                     -- ocp write data
      ScmdAccept_i:  in  std_logic;                   -- ocp command accept
      Sresp_i:       in  std_logic_vector(1 downto 0); -- ocp response
      Sdata_i:       in  regflit;                     -- ocp read data
      MrespAccept_o: out std_logic;
      busySlave:     in  std_logic;
      busyMaster:    out std_logic;
      ------------------------------ Switch Signals-----------------------------------------------
      data_inL:     in  regflit;
      rxL:          in  std_logic;
      ack_rxL:      out std_logic;
      data_outL:    out regflit;
      txL:          out std_logic;
      ack_txL:      in  std_logic);
end Master_OCP;

architecture Master_OCP of Master_OCP is

type send_state is (S_IDLE, SS1, SS2, SS3, SS4, SS5, SS6);
--where SS corresponds to send state while SR corresponds to receive state
signal CS: send_state;

type receive_state  is (S_IDLE, RR1, RR2, RR3, RR4, RR5, RR6, RR7);
--where SS corresponds to send state while SR corresponds to receive state
signal RR: receive_state;

------------------------------ SIGNALS DEFINITION-------------------------------------------------
-- TARGET:  It is the first  flit of the package and it has the local address (corrent nodo).
-- size: It's the secund flit and it represents the number of flits which compose the rest of the
pachage.
-- cmdOCP: This third flit defines the operation command, which can be: WRITE (1) or READ (2).
-- ADD_DATA: It is the fourth flit and it has the initial address of the memory.
-- N_DATA: This fifth  flit is used for representing the number of data that must be read.
-- SOURCE:And finally, the SOURCE signal (sixth flit), which defines the address of the source nodo.
-- CONT: This signal is used as a counter.  THAT'S IT!!!
-------------------------------------------------------------------------------------------------------
signal size,cmdOCP,cont,contR: regflit;
signal target,source: regmetadeflit;

begin

ack_rxL <= rxL when CS=S_IDLE or CS=SS1 or CS=SS2 else
           '1' when (rxL='1' and ScmdAccept_i='1') else
           '0';

    process(Mclk_i, Mreset_ni)
    begin
            if Mreset_ni= '1' then
                  CS <= S_IDLE;
                  Mcmd_o       <= (others=>'0');
                  Maddr_o     <= '0';
                  Mdata_o      <= (others=>'0');
                  target     <= (others=>'0');
                  size      <= (others=>'0');
                  cmdOCP    <= (others=>'0');
                  source     <= (others=>'0');
                  cont         <= (others=>'0');
            elsif Mclk_i'event and Mclk_i='1' then
                  case CS is
                        when S_IDLE =>
```

```
                              if rxL = '1' then
                                    source  <= data_inL(15 downto 8); -- receive the source/target
                                    target  <= data_inL(7 downto 0);
                                    CS      <= SS1;
                              else
                                    CS      <= S_IDLE;
                              end if;
                        when SS1 =>
                              if rxL = '1' then
                                    size   <= data_inL - 1; -- receive the size of the packet
                                    CS     <= SS2;
                              else
                                    CS        <= SS1;
                              end if;
                        when SS2 =>
                              if rxL = '1' then
                                    cmdOCP      <= data_inL; --receive the ocp command
                                    CS          <= SS3;
                              else
                                    CS        <= SS2;
                              end if;
                        when SS3 =>
                              if cmdOCP = x"09" then
                                    CS <= SS5;
                              elsif rxL = '1' then
                                    cont    <= cont + 1;
                                    CS          <= SS4;
                              else
                                    CS        <= SS3;
                              end if;
                        when SS4 =>
                              if cmdOCP = x"0001" then --Escrita
                                    Mcmd_o      <= WR;
                              elsif cmdOCP = x"0002" then --Leitura
                                    Mcmd_o      <= RD;
                              end if;
                              Mdata_o <= data_inL;
                              CS          <= SS4;
                              if ScmdAccept_i = '1' and cont /= size then  -- -2 despreza commandOCP
e source
                                    Mcmd_o   <= IDLE;
                                    CS <= SS3;
                              elsif ScmdAccept_i = '1' and cont = size then -- -2 despreza
commandOCP e source
                                    Mcmd_o   <= IDLE;
                                    cont         <= (others=>'0');
                                    CS     <= S_IDLE;
                              else
                                    CS  <= SS4;
                              end if;
                        when SS5 =>
                              if rxL = '1' then
                                    cont      <= cont + '1';
                                    CS        <= SS6;
                              else
                                    CS        <= SS5;
                              end if;
                        when SS6 =>
                              if rxL = '0' and cont /= size then
                                    CS        <= SS5;
                              elsif rxL = '0' and cont = size then
                                    cont      <= (others=>'0');
                                    CS        <= S_IDLE;
                              else
                                    CS        <= SS6;
                              end if;
                  end case;
            end if;
      end process;

-------------------------------------------------------------------------------------------------------
-----------------------------------RECEIVE HEADER STATES -----------------------------------------------
-------------------------------------------------------------------------------------------------------
    process(Mclk_i, Mreset_ni)
    begin
            if Mreset_ni= '1' then
                  RR <= S_IDLE;
                  busyMaster <= '0';
                  txL          <= '0';
                  data_outL    <= (others=>'0');
```

```
                contR <= (others=>'0');
            elsif Mclk_i'event and Mclk_i='0' then
                case RR is
                    when S_IDLE =>
                        busyMaster <= '0';
                        if busySlave='0' and Sresp_i=DVA then
                            busyMaster <= '1';
                            txL<='1';
                            data_outL <= target & source;
                            RR <= RR1;
                        else
                            RR <= S_IDLE;
                        end if;
                    when RR1 =>
                        if ack_txL = '1' then
                            txL      <= '0';
                            RR       <= RR2;
                            else
                            RR        <= RR1;
                        end if;
                    when RR2 =>
                        txL<='1';
                        data_outL <= size+1;
                        RR <= RR3;
                    when RR3 =>
                        if ack_txL = '1' then
                            txL       <= '0';
                            RR        <= RR4;
                            else
                            RR        <= RR3;
                        end if;
                    when RR4 =>
                        txL<='1';
                        data_outL <= (0=>'1',1=>'0',2=>'0',3=>'1',others=>'0') ;
                        RR <= RR5;
                    when RR5 =>
                        if ack_txL = '1' then
                            txL       <= '0';
                            contR <= (others=>'0');
                            RR        <= RR6;
                                else
                            RR        <= RR5;
                        end if;
                    when RR6 =>
                        if Sresp_i=DVA then
                            txL<='1';
                            data_outL <= Sdata_i;
                            contR <= contR + '1';
                            RR <= RR7;
                        else
                            RR <= RR6;
                        end if;
                    when RR7 =>
                        if ack_txL = '1' and contR/=size then
                            txL <= '0';
                            RR        <= RR6;
                        elsif ack_txL = '1' and contR=size then
                            txL <= '0';
                            contR <= (others=>'0');
                            RR        <= S_IDLE;
                            else
                            RR        <= RR7;
                        end if;
                end case;
            end if;
        end process;

        MrespAccept_o<='1' when ack_txL='1' and RR=RR7 else '0';
end Master_OCP;
```

ANEXO VI – CÓDIGO VHDL SLAVE_OCP.VHD

O código VHDL do par entidade-arquitetura da IR Slave-OCP:

```
library IEEE;
use ieee.STD_LOGIC_UNSIGNED.all;
use ieee.std_logic_1164.all;
use work.HermesPackage.all;

entity Slave_OCP is
port(
      ------------------------------OCP Signals-----------------------------------------------------
      Sclk_i:        in  std_logic;
      Sreset_ni:     in  std_logic;
      Mcmd_i:        in  reg3;                                  -- ocp command
      Maddr_i:       in  std_logic;                               -- ocp address
      Mdata_i:       in  regflit;                               -- ocp write data
      ScmdAccept_o:  out std_logic;                             -- ocp command accept
      Sresp_o:       out std_logic_vector(1 downto 0);          -- ocp response
      Sdata_o:       out regflit;                               -- ocp read data
      MrespAccept_i: in  std_logic;
      busyMaster:    in  std_logic;
      busySlave:     out std_logic;
      ------------------------------ Switch Signals-----------------------------------------------
      addressL:     in  regmetadeflit;
      data_inL:     in  regflit;
      rxL:          in  std_logic;
      ack_rxL:      out std_logic;
      data_outL:    out regflit;
      txL:          out std_logic;
      ack_txL:      in  std_logic
);
end Slave_OCP;

architecture Slave_OCP of Slave_OCP is

type ocp_state  is (S_IDLE, SS1, SS2, SS3, SS4, SS5, SS6);
type ocp_send  is (S_IDLE,SS1,SS2,SS3,SS4,SS5,SS6,SS7);
signal CS: ocp_state;
signal CSS: ocp_send;
signal size,sizeSend,cont,contSend,cmdOCP: regflit;
signal auxScmdAccept_o: std_logic;

begin

ScmdAccept_o <= auxScmdAccept_o;
-------------------------------------------------------------------------------------------------------
---- SEND
-------------------------------------------------------------------------------------------------------
      process(Sclk_i, Sreset_ni)
      begin
            if Sreset_ni='1' then
                  busySlave <= '0';
                  txL <= '0';
                  auxScmdAccept_o <= '0';
                  sizeSend <= (others=>'0');
                  data_outL <= (others=>'0');
                  contSend <= (others=>'0');
                  CSS <= S_IDLE;
            elsif Sclk_i'event and Sclk_i='1' then
                  case CSS is
                        when S_IDLE =>
                              busySlave <= '0';
                              auxScmdAccept_o <= '0';
                              if busyMaster='0' and Mcmd_i /= IDLE and  auxScmdAccept_o='0' then
                                    busySlave <= '1';
                                    -- Source & Target
                                    txL <= '1';
                                    data_outL <= addressL & Mdata_i((METADEFLIT-1) downto 0);
                                    CSS <= SS1;
                              else
                                    txL <= '0';
                                    CSS <= S_IDLE;
                              end if;
                        when SS1 =>   -- enquanto não responde fique neste estado
                              if ack_txL = '1' then
```

```
                              auxScmdAccept_o <= '1';
                              txL <= '0';
                              CSS <= SS2;
                        else
                              CSS <= SS1;
                        end if;
                  when SS2 =>
                        auxScmdAccept_o <= '0';
                        if Mcmd_i /= IDLE and auxScmdAccept_o='0' then
                              CSS <= SS3;
                        else
                              txL <= '0';
                              CSS <= SS2;
                        end if;
                  when SS3 =>   -- enquanto não responde fique neste estado
                        -- Size + 1 por causa do comandoOCP que deve ser inserido
                        txL <= '1';
                        sizeSend <= Mdata_i + '1';
                        data_outL <= Mdata_i + '1';
                        if ack_txL = '1' then
                              auxScmdAccept_o <= '1';
                              txL <= '0';
                              CSS <= SS4;
                        else
                              CSS <= SS3;
                        end if;
                  when SS4 =>
                        auxScmdAccept_o <= '0';
                        if Mcmd_i /= IDLE and auxScmdAccept_o='0' then -- recebendo o primeiro
dado do payload
                              CSS <= SS5;
                        else
                              txL <= '0';
                              CSS <= SS4;
                        end if;
                  when SS5 =>   -- enquanto não responde fique neste estado
                        -- comando OCP
                        txL <= '1';
                        data_outL <= x"000" & "0" & Mcmd_i;
                        if ack_txL = '1' then
                              contSend <= contSend + '1';
                              txL <= '0'; -- nao envia ScmdAccept_o pq commandOCP nao tem
origem do IP
                              CSS <= SS6;
                        else
                              CSS <= SS5;
                        end if;
                  when SS6 =>
                        auxScmdAccept_o <= '0';
                        if Mcmd_i /= IDLE and auxScmdAccept_o='0' then
                              contSend <= contSend + '1';
                              CSS <= SS7;
                        else
                              txL       <= '0';
                              CSS        <= SS6;
                        end if;
                  when SS7=>   -- enquanto não responde fique neste estado
                        -- Payload
                        txL <= '1';
                        data_outL <= Mdata_i;
                        if ack_txL = '1' and contSend=sizeSend then
                              auxScmdAccept_o <= '1';
                              txL       <= '0';
                              contSend <= (others=>'0');
                              CSS        <= S_IDLE;
                        elsif ack_txL = '1' and contSend/=sizeSend then
                              auxScmdAccept_o <= '1';
                              txL       <= '0';
                              CSS        <= SS6;
                        else
                              CSS        <= SS7;
                        end if;
                  end case;
            end if;
      end process;
-------------------------------------------------------------------------------------------
---- RECEIVE
-------------------------------------------------------------------------------------------------------
      ack_rxL <= rxL when CS=S_IDLE or CS=SS1 or CS=SS2 else
                  MrespAccept_i;
```

```
  ------------------------------------------------------------------------------------------
  ---------------------- OCP CONTROL MACHINE-----------------------------------------------
  ------------------------------------------------------------------------------------------
      process(Sclk_i, Sreset_ni)
      begin
        if Sreset_ni='1' then
            cont    <= (others=>'0');
            cmdOCP <= (others=>'0');
            size  <= (others=>'0');
            Sresp_o   <= NULO;
            Sdata_o   <= (others=>'0');
            CS <= S_IDLE;
            elsif Sclk_i'event and Sclk_i='1' then
                  case CS is
                        when S_IDLE =>
                              if rxL = '1' then
                                    CS      <= SS1;   -- souce/target nao armazena
                              else
                                    CS      <= S_IDLE;
                              end if;
                        when SS1 =>
                              if rxL = '1' then
                                    size <= data_inL - 1; -- size armazena
                                    CS      <= SS2;
                              else
                                    CS      <= SS1;
                              end if;
                        when SS2 =>
                              if rxL = '1' then
                                    cmdOCP <= data_inL;
                                    CS      <= SS3;  -- commandOCP nao armazena
                              else
                                    CS      <= SS2;
                              end if;
                        when SS3 =>
                              if cmdOCP /= x"09" then
                                    CS <= SS5;
                              elsif rxL = '1' then
                                    Sresp_o <= DVA; -- payload
                                    Sdata_o <= data_inL;
                                    cont <= cont + 1;
                                    CS <= SS4;
                              else
                                    CS <= SS3;
                              end if;
                        when SS4 =>
                              if MrespAccept_i = '1' and cont/=size then
                                    Sresp_o <= NULO;
                                    CS <= SS3;
                              elsif MrespAccept_i = '1' and cont=size then
                                    Sresp_o <= NULO;
                               cont    <= (others=>'0');
                                    CS <= S_IDLE;
                              else
                                    CS <= SS4;
                              end if;
                        when SS5 =>
                              if rxL = '1' then
                                    cont      <= cont + 1;
                                    CS        <= SS6;
                              else
                                    CS        <= SS5;
                              end if;
                        when SS6 =>
                              if rxL = '0' and cont /= size then
                                    CS        <= SS5;
                              elsif rxL = '0' and cont = size then
                                    cont      <= (others=>'0');
                                    CS        <= S_IDLE;
                              else
                                    CS        <= SS6;
                              end if;
             end case;
        end if;
    end process;

end Slave_OCP;
```

ANEXO VII – CÓDIGO VHDL MASTERSLAVE.VHD

O código VHDL do par entidade-arquitetura da IR MasterSlave OCP:

```
library IEEE;
use ieee.STD_LOGIC_UNSIGNED.all;
use ieee.std_logic_1164.all;
use work.HermesPackage.all;

--MasterSlave_OCP definition
entity MasterSlave_OCP is
port(
      ------------------------- Master OCP Signals-------------------------------------------------
      Mclk_i:        in  std_logic;
      Mreset_ni:     in  std_logic;
      Mcmd_o:        out reg3;                        -- ocp command
      Maddr_o:       out std_logic;                     -- ocp address
      Mdata_o:       out regflit;                     -- ocp write data
      ScmdAccept_i:  in  std_logic;                   -- ocp command accept
      Sresp_i:       in  std_logic_vector(1 downto 0); -- ocp response
      Sdata_i:       in  regflit;                     -- ocp read data
      MrespAccept_o: out std_logic;
      ------------------------- Slave OCP Signals-------------------------------------------------
      Sclk_i:        in  std_logic;
      Sreset_ni:     in  std_logic;
      Mcmd_i:        in  reg3;                        -- ocp command
      Maddr_i:       in  std_logic;                     -- ocp address
      Mdata_i:       in  regflit;                     -- ocp write data
      ScmdAccept_o:  out std_logic;                   -- ocp command accept
      Sresp_o:       out std_logic_vector(1 downto 0); -- ocp response
      Sdata_o:       out regflit;                     -- ocp read data
      MrespAccept_i: in  std_logic;
      ------------------------------ Switch Signals-----------------------------------------------
      addressL:      in  regmetadeflit;
      rxL:           in  std_logic;
      data_inL:      in  regflit;
      ack_rxL:       out std_logic;
      txL:           out std_logic;
      data_outL:     out regflit;
      ack_txL:       in  std_logic);
end MasterSlave_OCP;

architecture MasterSlave_OCP of MasterSlave_OCP is

signal StxL,MtxL,Sack_rxL,Mack_rxL: std_logic;
signal Sdata_outL,Mdata_outL: regflit;
signal busyMaster,busySlave: std_logic;

begin

      ack_rxL <= Mack_rxL or Sack_rxL;

      txL <= MtxL or StxL;
      data_outL <= Mdata_outL when busyMaster='1' else Sdata_outL;

-------------------------------------------------------------------------------------------------
--------------------------------- MASTER ----------------------------------------------------
-------------------------------------------------------------------------------------------------
      master: Entity work.Master_OCP (Master_OCP)
      port map(
            Mclk_i        => Mclk_i,
            Mreset_ni     => Mreset_ni,
            Mcmd_o        => Mcmd_o,
            Maddr_o       => Maddr_o,
            Mdata_o       => Mdata_o,
            ScmdAccept_i  => ScmdAccept_i,
            Sresp_i       => Sresp_i,
            Sdata_i       => Sdata_i,
            MrespAccept_o => MrespAccept_o,
            busyMaster    => busyMaster,
            busySlave     => busySlave,
            ------------------------
            data_inL      => data_inL,
            rxL           => rxL,
            ack_rxL       => Mack_rxL,
            data_outL     => Mdata_outL,
            txL           => MtxL,
```

```
            ack_txL       => ack_txL);

      slave: Entity work.Slave_OCP (Slave_OCP)
      port map(
                  Sclk_i        => Sclk_i,
                  Sreset_ni     => Sreset_ni,
                  Mcmd_i        => Mcmd_i,
                  Maddr_i       => Maddr_i,
                  Mdata_i       => Mdata_i,
                  ScmdAccept_o  => ScmdAccept_o,
                  Sresp_o       => Sresp_o,
                  Sdata_o       => Sdata_o,
                  MrespAccept_i => MrespAccept_i,
                  busyMaster    => busyMaster,
                  busySlave     => busySlave,
                  -------------------------
                  addressL      => addressL,
                  data_inL      => data_inL,
                  rxL           => rxL,
                  ack_rxL       => Sack_rxL,
                  data_outL     => Sdata_outL,
                  txL           => StxL,
                  ack_txL       => ack_txL);

end MasterSlave_OCP;
```

ANEXO VIII – O CÓDIGO VHDL NOC.VHD

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.HermesPackage.all;

entity NOC is
port(
      clock : in std_logic;
      reset : in std_logic;
      rxLocal : in  regNrot;
      data_inLocal : in  arrayNrot_regflit;
      ack_rxLocal : out regNrot;
      txLocal : out regNrot;
      data_outLocal : out arrayNrot_regflit;
      ack_txLocal : in  regNrot);
end NOC;

architecture NOC of NOC is

      signal rxN0000, rxN0100 : regNport;
      signal data_inN0000, data_inN0100 : arrayNport_regflit;
      signal ack_rxN0000, ack_rxN0100 : regNport;
      signal txN0000, txN0100 : regNport;
      signal data_outN0000, data_outN0100 : arrayNport_regflit;
      signal ack_txN0000, ack_txN0100 : regNport;
      signal rxN0001, rxN0101 : regNport;
      signal data_inN0001, data_inN0101 : arrayNport_regflit;
      signal ack_rxN0001, ack_rxN0101 : regNport;
      signal txN0001, txN0101 : regNport;
      signal data_outN0001, data_outN0101 : arrayNport_regflit;
      signal ack_txN0001, ack_txN0101 : regNport;
begin

      Chave0000 : Entity work.ChaveTL(ChaveTL)
      generic map( address => ADDRESSN0000 )
      port map(
            clock => clock,
            reset => reset,
            rx => rxN0000,
            data_in => data_inN0000,
            ack_rx => ack_rxN0000,
            tx => txN0000,
            data_out => data_outN0000,
            ack_tx => ack_txN0000
      );

      Chave0100 : Entity work.ChaveTR(ChaveTR)
      generic map( address => ADDRESSN0100 )
      port map(
            clock => clock,
            reset => reset,
            rx => rxN0100,
            data_in => data_inN0100,
            ack_rx => ack_rxN0100,
            tx => txN0100,
            data_out => data_outN0100,
            ack_tx => ack_txN0100
      );

      Chave0001 : Entity work.ChaveBL(ChaveBL)
      generic map( address => ADDRESSN0001 )
      port map(
            clock => clock,
            reset => reset,
            rx => rxN0001,
            data_in => data_inN0001,
            ack_rx => ack_rxN0001,
            tx => txN0001,
            data_out => data_outN0001,
            ack_tx => ack_txN0001
      );

      Chave0101 : Entity work.ChaveBR(ChaveBR)
      generic map( address => ADDRESSN0101 )
      port map(
            clock => clock,
```

```
      reset => reset,
      rx => rxN0101,
      data_in => data_inN0101,
      ack_rx => ack_rxN0101,
      tx => txN0101,
      data_out => data_outN0101,
      ack_tx => ack_txN0101
);

-- entradas do Switch0000
data_inN0000(0)<=data_outN0100(1);
rxN0000(0)<=txN0100(1);
ack_txN0000(0)<=ack_rxN0100(1);
data_inN0000(1)<=(others=>'0');
rxN0000(1)<='0';
ack_txN0000(1)<='0';
data_inN0000(2)<=(others=>'0');
rxN0000(2)<='0';
ack_txN0000(2)<='0';
data_inN0000(3)<=data_outN0001(2);
rxN0000(3)<=txN0001(2);
ack_txN0000(3)<=ack_rxN0001(2);
data_inN0000(4)<=data_inLocal(N0000);
rxN0000(4)<=rxLocal(N0000);
ack_txN0000(4)<=ack_txLocal(N0000);
ack_rxLocal(N0000)<=ack_rxN0000(4);
data_outLocal(N0000)<=data_outN0000(4);
txLocal(N0000)<=txN0000(4);

-- entradas do Switch0100
data_inN0100(0)<=(others=>'0');
rxN0100(0)<='0';
ack_txN0100(0)<='0';
data_inN0100(1)<=data_outN0000(0);
rxN0100(1)<=txN0000(0);
ack_txN0100(1)<=ack_rxN0000(0);
data_inN0100(2)<=(others=>'0');
rxN0100(2)<='0';
ack_txN0100(2)<='0';
data_inN0100(3)<=data_outN0101(2);
rxN0100(3)<=txN0101(2);
ack_txN0100(3)<=ack_rxN0101(2);
data_inN0100(4)<=data_inLocal(N0100);
rxN0100(4)<=rxLocal(N0100);
ack_txN0100(4)<=ack_txLocal(N0100);
ack_rxLocal(N0100)<=ack_rxN0100(4);
data_outLocal(N0100)<=data_outN0100(4);
txLocal(N0100)<=txN0100(4);

-- entradas do Switch0001
data_inN0001(0)<=data_outN0101(1);
rxN0001(0)<=txN0101(1);
ack_txN0001(0)<=ack_rxN0101(1);
data_inN0001(1)<=(others=>'0');
rxN0001(1)<='0';
ack_txN0001(1)<='0';
data_inN0001(2)<=data_outN0000(3);
rxN0001(2)<=txN0000(3);
ack_txN0001(2)<=ack_rxN0000(3);
data_inN0001(3)<=(others=>'0');
rxN0001(3)<='0';
ack_txN0001(3)<='0';
data_inN0001(4)<=data_inLocal(N0001);
rxN0001(4)<=rxLocal(N0001);
ack_txN0001(4)<=ack_txLocal(N0001);
ack_rxLocal(N0001)<=ack_rxN0001(4);
data_outLocal(N0001)<=data_outN0001(4);
txLocal(N0001)<=txN0001(4);

-- entradas da Switch0101
data_inN0101(0)<=(others=>'0');
rxN0101(0)<='0';
ack_txN0101(0)<='0';
data_inN0101(1)<=data_outN0001(0);
rxN0101(1)<=txN0001(0);
ack_txN0101(1)<=ack_rxN0001(0);
data_inN0101(2)<=data_outN0100(3);
rxN0101(2)<=txN0100(3);
ack_txN0101(2)<=ack_rxN0100(3);
data_inN0101(3)<=(others=>'0');
```

```
      rxN0101(3)<='0';
      ack_txN0101(3)<='0';
      data_inN0101(4)<=data_inLocal(N0101);
      rxN0101(4)<=rxLocal(N0101);
      ack_txN0101(4)<=ack_txLocal(N0101);
      ack_rxLocal(N0101)<=ack_rxN0101(4);
      data_outLocal(N0101)<=data_outN0101(4);
      txLocal(N0101)<=txN0101(4);

end NOC;
```